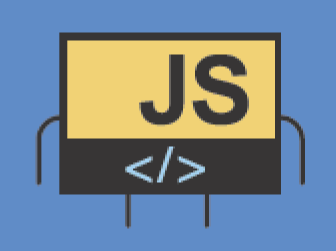


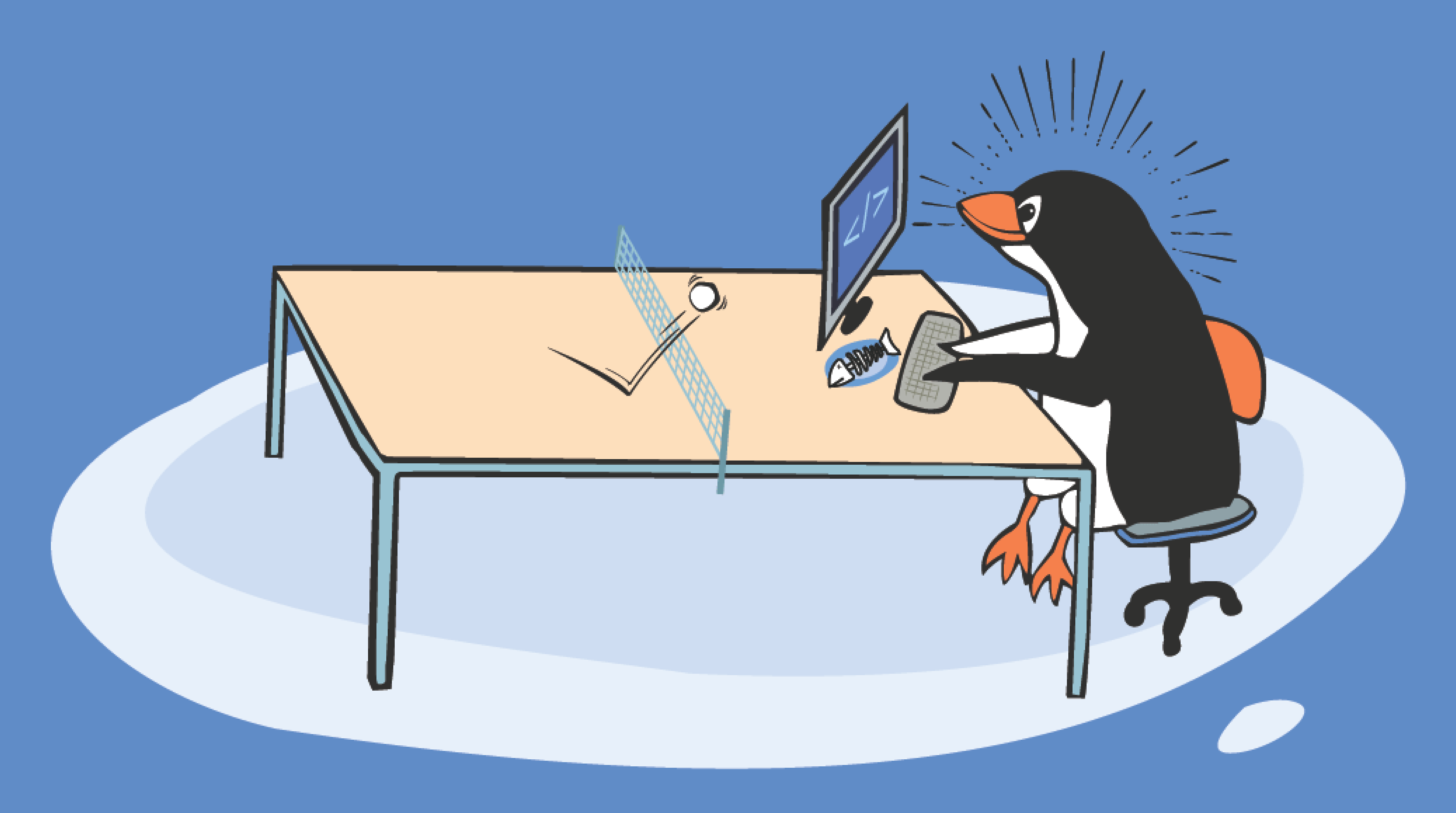
编程新手无痛学JavaScript

ES6新特性 / 活用语法 / 编程技巧 / 编程练习 / 项目实践

现代JavaScript编程

经典范例与实践技巧

张益珲　吕　远　编著

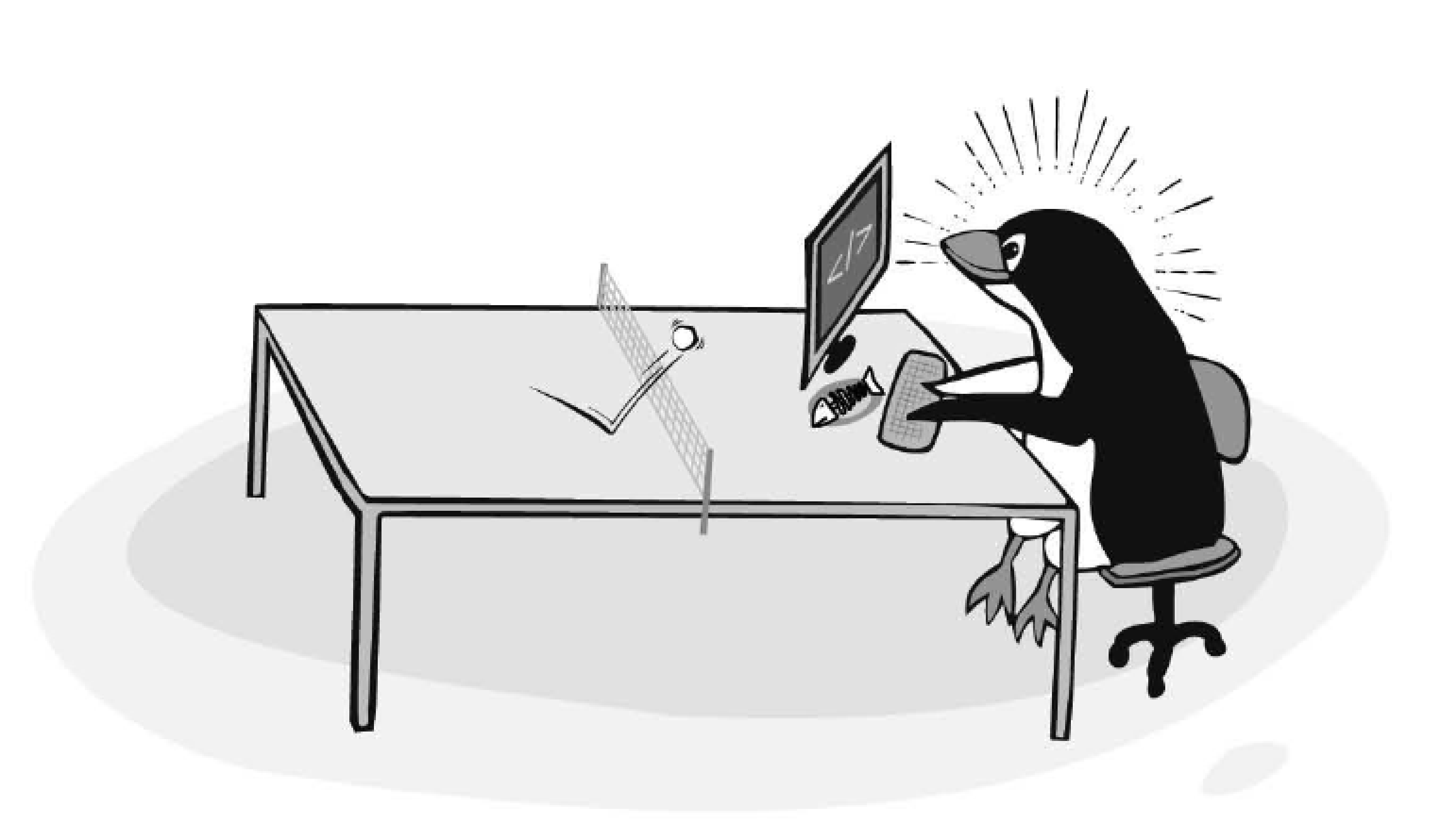


清華大學出版社

现代JavaScript编程

经典范例与实践技巧

张益珲　吕　远　编著



清華大学出版社
北　京

内 容 简 介

JavaScript 作为流行的脚本语言，其应用方向也从开始只作为网页脚本，到现在可以做网页应用程序、React Native 跨平台移动端应用、后端服务等。作为现代开发者，JavaScript 无疑成为必须掌握的一门技能。

本书从 JavaScript 的基本语法、函数与对象、高级特性到设计模式、HTML DOM/BOM 对 JavaScript 的语法、编程思想以及应用进行了全面的讲解。本书的特色是介绍了 JavaScript ES 6 的新语法，将复杂的 JavaScript 语言划分成 100 多个主题进行讲解，并在各章设计了大量的编程练习，在本书的最后还设计了两个实用的小项目，旨在帮助读者开发出自己的应用程序。

本书适合想快速学习 JavaScript 的编程初学者、学生以及对编程感兴趣的人员。

图书在版编目（CIP）数据

现代 JavaScript 编程：经典范例与实践技巧/张益珲，吕远编著.—北京：清华大学出版社，2018
ISBN 978-7-302-50638-6

Ⅰ. ①现… Ⅱ. ①张… ②吕… Ⅲ. ①JAVA 语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2018）第 156485 号

责任编辑： 王金柱
封面设计： 王　翔
责任校对： 闫秀华
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者： 北京鑫海金澳胶印有限公司
经　　销： 全国新华书店
开　　本： 180mm×230mm　　**印　　张：** 15.5　　**字　　数：** 347 千字
版　　次： 2018 年 8 月第 1 版　　**印　　次：** 2018 年 8 月第 1 次印刷
定　　价： 59.00 元

产品编号：076915-01

前　　言

当你拿到这本书时，首先要感谢你在众多编程书籍中选择本书。我想要告诉你的是，这是一本讲解 JavaScript 语法的工具书，但却不仅仅是一本工具书。除了学习 JavaScript 外，本书会更多地锻炼你的编程思维，提高你的程序理解与设计能力。如果你是一个编程界的小白，那么恭喜你，本书对你再合适不过了。

很长一段时间，JavaScript 语言都被一些开发者戏称为“玩具语言”。的确，在移动设备未普及、网络传输速度不够快的时代，JavaScript 更多的是用来进行网页的部分动态展示和动画开发。和强大的 Java、C++等编译型语言相比，JavaScript 的确简单得多。然而，这并非表示 JavaScript 本身不够强大，只是还没有完全展现出来而已。

随着移动端设备的普及与无线网速度越来越快，移动应用逐渐代替传统的桌面应用，单页面网页应用与响应式移动端应用更是得到飞速的发展，现在你可以十分容易地在云上进行协同办公，可以在毫无感知的情况下更新自己的应用程序，获得更优质的服务，这些都要归功于 JavaScript，本书将带你领略 JavaScript 的美妙。

本书在结构上分为 8 个章节，总体上遵循由易到难的安排方式。

第 1 章为快速体验 JavaScript，本章将向你介绍一些 JavaScript 的基本编码规则、JavaScript 的语法特点以及 JavaScript 一些简单的概念。并且在本章中将教你配置 JavaScript 运行环境以及调试 JavaScript 代码。本章的安排主要是让你在学习之前可以简单认识一下 JavaScript 这门语言，如果你以前从未接触过它，相信会使你耳目一新。

第 2 章为 ECMAScript 的语法世界，你会在本章学习到变量、作用域、数据类型、对象、运算符和类型转换的相关知识。通过本章的学习，你能够掌握使用 JavaScript 编写简单的运算程序，能够用 JavaScript 处理简单的逻辑问题。

第 3 章为 ECMAScript 流程控制和函数，有了流程语句，你的程序便有了一定程度上的智能。函数则更进一步，使程序可以拆分成一个一个的功能模块，理论上讲，学习完本章，你就可以使用 JavaScript 解决大部分编程问题。

第 4 章为 ECMAScript 面向对象编程。面向对象是人类在编程界的一大发明，也是现代编程领域中流行的编程方式。有了面向对象，程序才真正地变成了一个世界，编程也真正地变成了一种艺术。巧的是，JavaScript 是一种完全的面向对象语言，但是其又

不是传统意义上的基于类的面向对象语言，这将十分有趣，相信本章的内容一定会让你兴趣盎然。

第 5 章为 ECMAScript 的高级特性，其中很多是 ES6 中新增的特性。ES6 使得 JavaScript 的功能有了极大的提升，本章中介绍的解构赋值、箭头函数、代理对象、承诺对象、状态机对象等都会成为你使用 JavaScript 编程的“绝世好剑”。

第 6 章为 JavaScript 常用设计模式，虽然这些设计模式都是通过 JavaScript 进行实现和演示的，但是它们和 JavaScript 并没有特别大的关系。在编程领域，设计模式的思想是通用的，甚至和你生活中的思考方式也是通用的。因此，本章将是你的一场思维盛宴。

第 7 章为 JavaScript HTML DOM/BOM，主要介绍 HTML DOM 和 HTML BOM 的相关知识，因为 JavaScript 最简单的应用就是操作 HTML DOM 和 BOM。

第 8 章为 JavaScript 项目实战，实战是检测你学习成果的一种好方法，并且在实际使用中，你也能更深入地理解所学习到的知识。

希望本书可以帮助你达到学习目标，如果你想获取 JavaScript 编程的相关教学视频，可以通过以下网址访问我的网络课程：

https://edu.csdn.net/lecturer/course_list

如果你在学习过程中遇到问题或者发现本书的遗漏或错误之处，可以随时与我联系，我的 QQ 是：316045346。当然，在本书出版的过程中，我和出版社的编辑以及所有校对和整理本书的朋友都付出了很多汗水，尽量保证可以尽善尽美地让它呈现在你的面前。最后，感谢王金柱编辑在本书编写过程中提出的宝贵意见和在修订过程中的辛苦工作，感谢吕远同学提供的校稿服务，感谢其他所有为本书出版付出汗水的人。如果本书可以给你带来提高与帮助，我们的辛苦就会更有价值。

珲少

2018 年 5 月 24 日 上海

目　　录

第 1 章　快速体验 JavaScript …… 1
1-1　使用 Chrome 运行 JavaScript 代码 …… 1
1-2　JavaScript 的变量 …… 2
1-3　不同数据类型之间的运算 …… 2
1-4　分支语句 …… 3
1-5　循环语句 …… 4
1-6　函数 …… 5
1-7　理解对象 …… 5
1-8　数组对象的使用 …… 6
1-9　编写闰年判断小程序 …… 7
1-10　使用 Sublime Text 来编写 JavaScript 代码 …… 8
1-11　JavaScript 的语法特点 …… 16
1-12　编程练习 …… 17
第 2 章　ECMAScript 的语法世界 …… 20
2-1　理解变量 …… 20
2-2　变量的命名 …… 21
2-3　变量提升 …… 24
2-4　块级作用域 …… 25
2-5　ECMAScript 中的数据类型 …… 27
2-6　再看 const 关键字 …… 28
2-7　Undefined 与 Null …… 29
2-8　关于 Boolean 类型 …… 30
2-9　关于 Number 类型 …… 31
2-10　关于 String 类型 …… 32
2-11　对象简介 …… 33
2-12　算术运算符 …… 35
2-13　赋值运算符 …… 39
2-14　关系运算符 …… 40
2-15　逻辑运算符 …… 42
2-16　位运算符 …… 44
2-17　自增与自减运算符 …… 49
2-18　条件运算符 …… 50
2-19　逗号运算符与 delete 运算符 …… 50

2-20 关于运算符的优先级与结合性……51
2-21 隐式类型转换……53
2-22 编程练习……55

第 3 章 ECMAScript 流程控制与函数……59

3-1 if-else 分支结构……59
3-2 switch-case 分支结构……60
3-3 while 循环结构……62
3-4 for 循环结构……63
3-5 关于 for-in 与 for-of 结构……64
3-6 break 中断语句……66
3-7 continue 中断语句……68
3-8 异常抛出语句 throw……69
3-9 对异常进行捕获处理……71
3-10 传递异常……73
3-11 使用函数语句定义函数……76
3-12 使用函数表达式定义函数……77
3-13 使用 Function 构造器定义函数……78
3-14 立即执行函数……79
3-15 编程练习……80

第 4 章 ECMAScript 面向对象编程……85

4-1 创建对象……86
4-2 设置对象的属性和行为……87
4-3 内置 Number 对象……88
4-4 Number 对象与 Number 数值……90
4-5 内置 String 对象……91
4-6 与 HTML 相关的 String 方法……93
4-7 内置 Boolean 对象……94
4-8 内置 Array 对象……95
4-9 内置 Date 对象……99
4-10 内置 Math 对象……103
4-11 内置 RegExp 正则表达式对象……105
4-12 内置 Function 对象……109
4-13 内置 Object 对象……112
4-14 进行对象属性的配置……112
4-15 Object 函数对象常用方法……114
4-16 Object 实例对象常用方法……119
4-17 面向对象编程中的几个重要概念……120

4-18　用工厂方法模拟类 …… 121
4-19　使用构造方法模拟类 …… 122
4-20　使用 Object 函数对象的 create 方法模拟类 …… 123
4-21　使用封装法模拟类 …… 123
4-22　使用对象冒充的方式实现继承 …… 124
4-23　使用原型链的方式实现继承 …… 126
4-24　使用混合模式实现继承 …… 128
4-25　编程练习 …… 129

第 5 章　ECMAScript 的高级特性 …… 133

5-1　数组的解构赋值 …… 133
5-2　对象的解构赋值 …… 135
5-3　字符串与函数参数的解构赋值 …… 137
5-4　用解构赋值交换变量的值 …… 138
5-5　箭头函数的基本用法 …… 138
5-6　箭头函数中 this 的固化 …… 139
5-7　Set 集合结构 …… 141
5-8　Map 字典结构 …… 144
5-9　使用 Proxy 代理对对象的属性读写进行拦截 …… 146
5-10　Proxy 代理处理器支持的拦截操作 …… 147
5-11　使用 Promise 承诺对象 …… 150
5-12　建立 Promise 任务链 …… 152
5-13　进行 Promise 对象组合 …… 153
5-14　Generator 函数应用 …… 155
5-15　Generator 任务参数的传递 …… 158
5-16　使用 class 定义类 …… 159
5-17　使用 class 实现类的继承 …… 160
5-18　认识 JSON 数据格式 …… 162
5-19　使用 JSON 对象 …… 163
5-20　认识 Symbol …… 165
5-21　注册全局的 Symbol 符号 …… 166
5-22　迭代器 Symbol …… 167
5-23　正则表达式符号 …… 167
5-24　使用 export 进行模块的导出 …… 168
5-25　使用 import 进行模块的导入 …… 169
5-26　编程练习 …… 170

第 6 章　JavaScript 常用设计模式 …… 173

6-1　工厂设计模式 …… 173

6-2 单例设计模式 …… 176
6-3 建造者设计模式 …… 177
6-4 适配器设计模式 …… 180
6-5 装饰器设计模式 …… 181
6-6 外观设计模式 …… 182
6-7 享元设计模式 …… 184
6-8 代理设计模式 …… 186
6-9 责任链设计模式 …… 187
6-10 命令设计模式 …… 189
6-11 迭代器设计模式 …… 190
6-12 备忘录设计模式 …… 191
6-13 观察者设计模式 …… 193
6-14 编程练习 …… 194

第 7 章 JavaScript HTML DOM/BOM …… 197

7-1 创建学习模板 …… 197
7-2 几个重要概念 …… 199
7-3 Document 文档对象 …… 199
7-4 Element 节点对象 …… 202
7-5 Attribute 属性对象 …… 206
7-6 用户事件 …… 208
7-7 Event 事件对象 …… 209
7-8 关于事件传递 …… 210
7-9 简单的轮播广告 …… 211
7-10 Window 窗口对象 …… 213
7-11 Navigator 导航对象 …… 216
7-12 Screen 屏幕对象 …… 217
7-13 History 历史对象 …… 217
7-14 Location 地址对象 …… 218
7-15 编程练习 …… 218

第 8 章 JavaScript 项目实战 …… 222

8-1 项目一：编写一个简易网页时钟 …… 222
8-1-1 关于 Canvas 标签 …… 223
8-1-2 制作简易网页时钟 …… 225
8-2 项目二：编写网页笑话阅读器 …… 229
8-2-1 通过互联网获取免费的应用数据 …… 229
8-2-2 关于 AJAX …… 232
8-2-3 代码实现 …… 234

第 1 章

快速体验 JavaScript

如果你是初学者，本章将告诉你如何编写JavaScript程序，本章分别讲解了在浏览器和编辑器中编写JavaScript代码的方法，变量、条件、循环、对象等基本概念以及JavaScript语言的特点，让你对JavaScript这门语言有个概览并且能够简单地应用。

1-1 使用 Chrome 运行 JavaScript 代码

JavaScript最大的用武之地在于浏览器，浏览器也提供了越来越丰富的JavaScript调试工具，不夸张地讲，你完全可以使用浏览器工具进行JavaScript开发。现在，就请打开谷歌浏览器Chrome，使用快捷键Command+Option+J就可以直接打开调试模式，如图1-1所示。

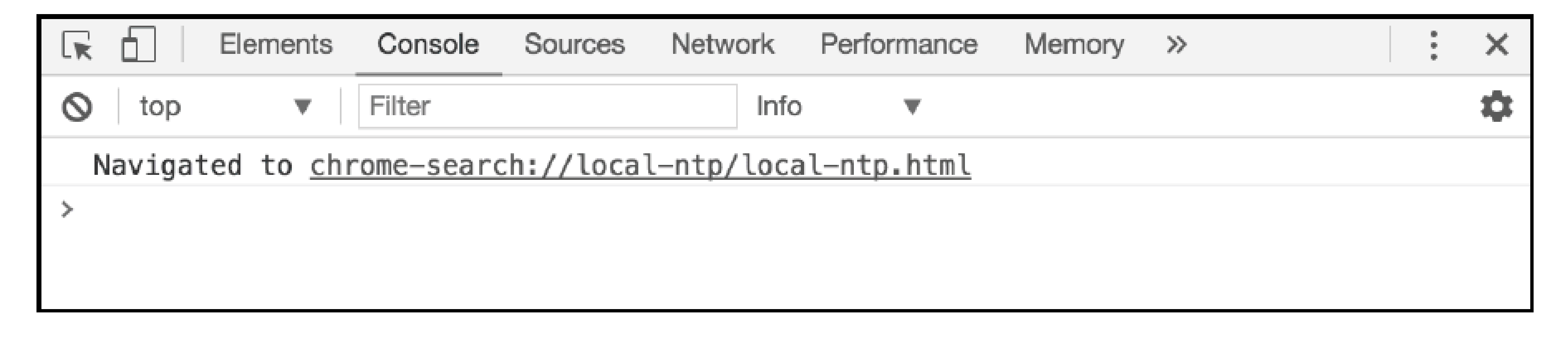


图 1-1 Chrome 浏览器的调试工具箱

在Console区，键入JavaScript代码后，可以直接在当前浏览器页面中运行，例如，在其中编写如下代码后按回车键：

```
document.write("<h1>欢迎你学习本书</h1>");
```

此时，浏览器的界面会显示如图1-2所示的信息。

欢迎您学习本书

图 1-2　浏览器界面

1-2　JavaScript 的变量

变量是存储信息的容器。在JavaScript中，你总是需要使用变量来进行值的存储和传递。JavaScript中的变量是无类型的，这也是JavaScript语言的一大特点，因为变量无类型，开发者可以用同一个变量存储各种各样的值。var关键字用来声明变量，在ES6（ECMAScript 6）中提供了let关键字，其声明的变量会受到块级作用域的影响，应用也更加广泛。请看下面的例子：

```
//使用 var 声明变量
var name;
//为变量赋值
name = "Jaki";
//使用 let 定义变量
let age = 25;
```

1-3　不同数据类型之间的运算

JavaScript中提供了常用的数学运算符，如“+”“-”“*”“/”“%”等，还提供了众多逻辑运算符、复合运算符、位运算符和比较运算符。如果需要更高级的数学运算，你也可以使用JavaScript中的Math对象，这是一个专门用来进行数学计算以及提供数学常数的对象。需要注意，在JavaScript中经常会出现不同数据类型之间的运算操作，例如数值和字符串进行加法运算时，会将拼接成的完整字符串作为结果。数学运算符的用法如下：

```
//数值的加、减、乘、除、取余运算
var a=1;
```

```
var b=3;
var c = a+b;            //4
var d = a*b;            //3
var e = b/a;            //3
var f = a%3;            //1
//数值和字符串+运算
var g = a+"hello";            //"1hello"
//字符串+运算
var h = "hello"+"world";            //"helloworld"
```

1-4　分支语句

对于编程语言，函数是最小的功能单元，语句是最小的处理单元。语句决定了程序执行的结构。分支语句是编程中十分重要的一种结构，如果没有分支语句，程序便无智能性可言。和C语言类似，JavaScript中的常用分支语句有if、if-else、switch-case等。示例如下：

```
let a = 1;
let b = 2;
if (a>b) {
    console.log("a>b");
}else if(a<b){
    console.log("a<b");
}else{
    console.log("a==b");
}
switch(a) {
    case 1:{
        console.log("a==1");
    }
    break;
    case 2:{
        console.log("a==2");
    }
    break;
    default:{
        console.log("a");
    }
```

```
}
//将打印 a<b   a==1
```

console.log()是浏览器中的打印函数，它会将结果输出到Console控制台。

分支语句给了程序做选择的能力，笔者建议if语句的嵌套最好不要超过3层，过多的if嵌套会使代码读起来非常难于理解。毕竟代码的真正意义是：给别人看，顺便可以在机器上运行。

1-5 循环语句

计算机和人类相比，最大的优势在于重复地做某件事情。对于程序来说，大量重复某些计算也十分必要。例如，统计公司中1000个员工的薪资，笔者相信没有任何一个软件工程师会将同样的统计代码编写1000遍。JavaScript中也提供了几种基础的循环语句，例如while语句、for语句等。对于对象，JavaScript中还提供了遍历对象属性的for-in方法。某些特殊对象（例如数组）中封装了许多迭代方法，这些会在后面的章节具体介绍。在JavaScript中，几种简单的循环结构示例如下：

```
//for 循环
var a=0;
for (var i = 0; i < 10; i++) {
    a=a+1;
}
console.log(a);        //10

//while 循环
while(a>0){
    a=a-1;
}
console.log(a);        //0

//do-while 循环
do{
    a=a+1;
}while(a<10);
console.log(a);        //10
```

1-6　函　　数

函数是JavaScript中的“第一等公民”，函数在JavaScript中的重要性和高级特性后面会有专门的章节进行系统地介绍。在JavaScript中，函数以function关键字定义，当然在ES6标准中，也可以使用箭头函数。函数是实现某一功能的代码单元，例如模拟加法运算，代码如下：

```
function add(a,b) {
    return a+b;
}
console.log(add(1,2));
```

你也可以将函数赋值给变量，将变量作为函数进行调用，这种函数通常被称为匿名函数，代码如下：

```
let addFunc = function(a,b){
    return a+b;
}
console.log(addFunc(1,3));
```

如果使用箭头函数的写法，示例代码如下：

```
let newAdd = (a,b)=>{
    return a+b;
}
console.log(newAdd(1,4));
```

1-7　理解对象

JavaScript是一种面向对象语言，对象是JavaScript的核心。你可以简单地将JavaScript中的对象理解为键值映射，在其他语言中，这种数据结构通常也叫作Dictionary或Map。JavaScript中内置了许多对象，例如Number数值对象、String字符串对象等。你也可以创建自定义的对象，代码如下：

```
var people = {
    name:'jaki',
    age:25
}
```

上面的代码定义了一个“人”对象，其中有两个属性，分别表示这个人的姓名和年龄。对象中除了可以定义属性来存储对象的内容外，也可以定义方法来描述对象的行为，例如：

```
var people = {
    name:'jaki',
    age:25,
    sayHi:()=>{
        console.log("Hello World");
    }
}
```

无论是访问对象的属性还是调用对象的方法，在JavaScript中都可以使用点语法，例如：

```
console.log(people.name);
people.sayHi();
```

如果你有其他语言的面向对象编程基础，可能会对上面的代码有些疑惑，JavaScript中的对象并不依赖于类，后面章节会有更深入的专题介绍。

1-8　数组对象的使用

数组在JavaScript中也是一种对象，其是JavaScript内置的Array对象。在实际开发中，数组的应用非常广泛。JavaScript中的Array对象存放数据的类型也十分自由，并不需要全部一致，例如：

```
var array = [1,"one","一"];
console.log(array[0]);        //1
console.log(array[1]);        //one
```

JavaScript中的数组在创建时也没有严格的长度限制，你可以在需要的时候任意增长或缩短数组的长度。例如，使用push方法可以在数组的末尾追加元素，使用pop方法也可以将数组末尾的元素删除。

```
array.pop();
array.push(2);
```

1-9　编写闰年判断小程序

通过上面的概览，你应该已经可以使用JavaScript结合一点HTML知识来编写简单的网页小程序了（例如判断某个年份是否为闰年）。关于闰年的知识，笔者相信你在小学数学课上就学习过。闰年的判断条件为：可以被4整除但是不能被100整除，或者可以直接被400整除的年份。下面就来编写这样一个小程序。

新建一个HTML文件，在其中编写如下代码：

```
<!DOCTYPE html>
<html>
<head>
	<title>闰年判断小程序</title>
	<script type="text/javascript">
		function check(){
			let value = document.getElementsByName("input")[0].value;
			if (!Number(value)) {
				alert("请你输入正确的年份");
				return;
			}
			if (Number(value)%4===0 && Number(value)%100!==0) {
				alert(value+"是闰年!");
			}else if (Number(value)%400===0) {
				alert(value+"是闰年!");
			}else{
				alert(value+"是平年!");
			}
		}
	</script>
	<style type="text/css">
		h1 {
			text-align: center;
		}
		div {
			text-align: center;
		}
		button {
```

```
                    margin-left: 30px;
               }
          </style>
     </head>
     <body>
          <h1>闰年判断小程序</h1>
          <div><input type="text" name="input" /><button onclick="check()">检验</button></div>
     </body>
     </html>
```

上面使用到的getElementsByName方法是document对象内置的方法，可以通过name来获取一组标签对象。

在浏览器中打开此文件，输入年份，即可进行验证，效果如图1-3所示。



图 1-3　闰年判断小程序

1-10　使用 Sublime Text 来编写 JavaScript 代码

Chrome浏览器提供的开发者工具虽然强大，但是只适用于已完成项目的检查与调试，无法用来进行完整项目的开发。可以编写JavaScript代码的编辑器十分多，例如专门开发大型Web项目的WebStorm、小巧的用于开发移动端网页的HBuilder、通用编辑器Sublime Text等工具。对于JavaScript语法部分的学习，强烈建议使用Sublime Text工具。首先，其拥有轻量级、占内存极小、运行极快的特点。其次，Sublime Text有大量插件支持，能够很好地提供代码高亮、智能补全、代码格式化等高级开发工具所提供的功能。最重要的是，Sublime Text可以配置编译系统，有了它，我们就不再需要依赖浏览器来进行JavaScript代码的运行与调试了。

笔者推荐使用Sublime Text 3.0版本，本书的示例操作和命令都是以3.0为准的。

下载Sublime Text最新版本的网址：http://www.sublimetext.com/。

下载安装完成Sublime Text后，其已经自带了代码高亮的功能，可以进行JavaScript代码的编写，但是并没有自动补全、代码格式化与运行JavaScript代码的功能，这里会为大家一一解决，现在我们将把Sublime Text武装成一款强大的JavaScript编辑器。

1. 安装 Sublime Text 插件管理器 PackageControl

Sublime Text的插件十分丰富，但是快速找到并安装所需要的插件很不容易，如果你有开发iOS软件的经验，一定知道CocoaPods第三方库管理工具。对应Sublime Text，PackageControl就是一款很好的插件管理器。

在Sublime Text中有两种方式可以进行PackageControl插件的安装：第一种方式是直接在Sublime Text的控制台键入如下代码，之后按Enter键来进行PackageControl工具的安装：

```
import urllib.request,os; pf = 'Package Control.sublime-package';
    ipp = sublime.installed_packages_path();
urllib.request.install_opener( urllib.request.build_opener( urllib.request.ProxyHandler()) );
open(os.path.join(ipp, pf), 'wb').write(urllib.request.urlopen( 'http://sublime.wbond.net/' +
pf.replace(' ','%20')).read())
```

在Sublime Text中使用Control+` 的方式可以直接打开控制台，将上面的代码粘贴进去并按Enter键进行安装，如图1-4所示。

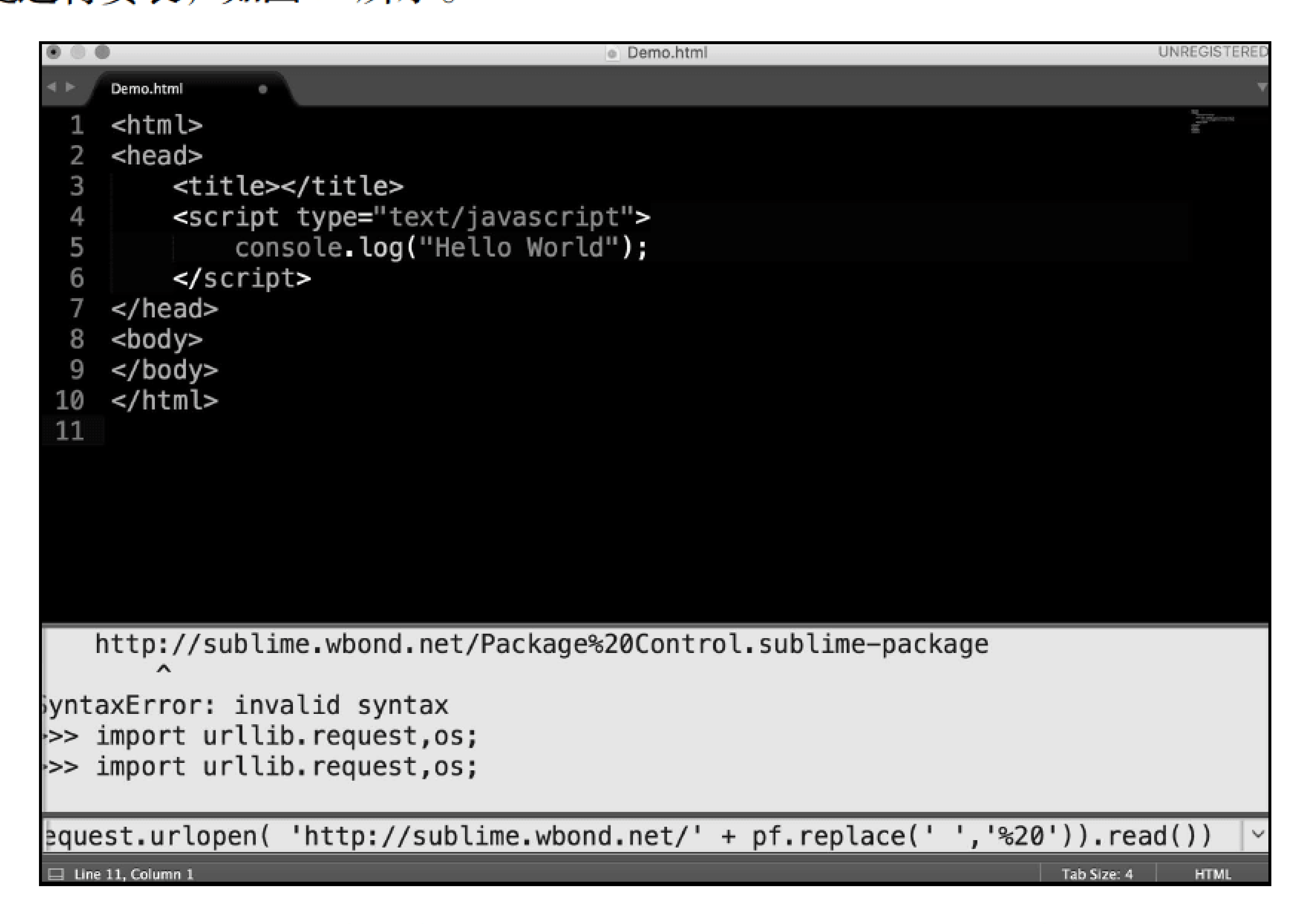


图 1-4　使用代码的方式进行 PackageControl 工具的安装

第二种安装方式是直接下载，手动安装。由于网络与Sublime Text版本更新不可控，使用

代码进行PackageControl工具的安装不一定会成功，可以直接下载PackageControl进行安装，PackageControl工具下载地址：http://sublime.wbond.net/Package%20Control.sublime-package。

下载完成后，如图1-5所示，打开Sublime Text工具的插件目录：Preferences→Browse Packages。

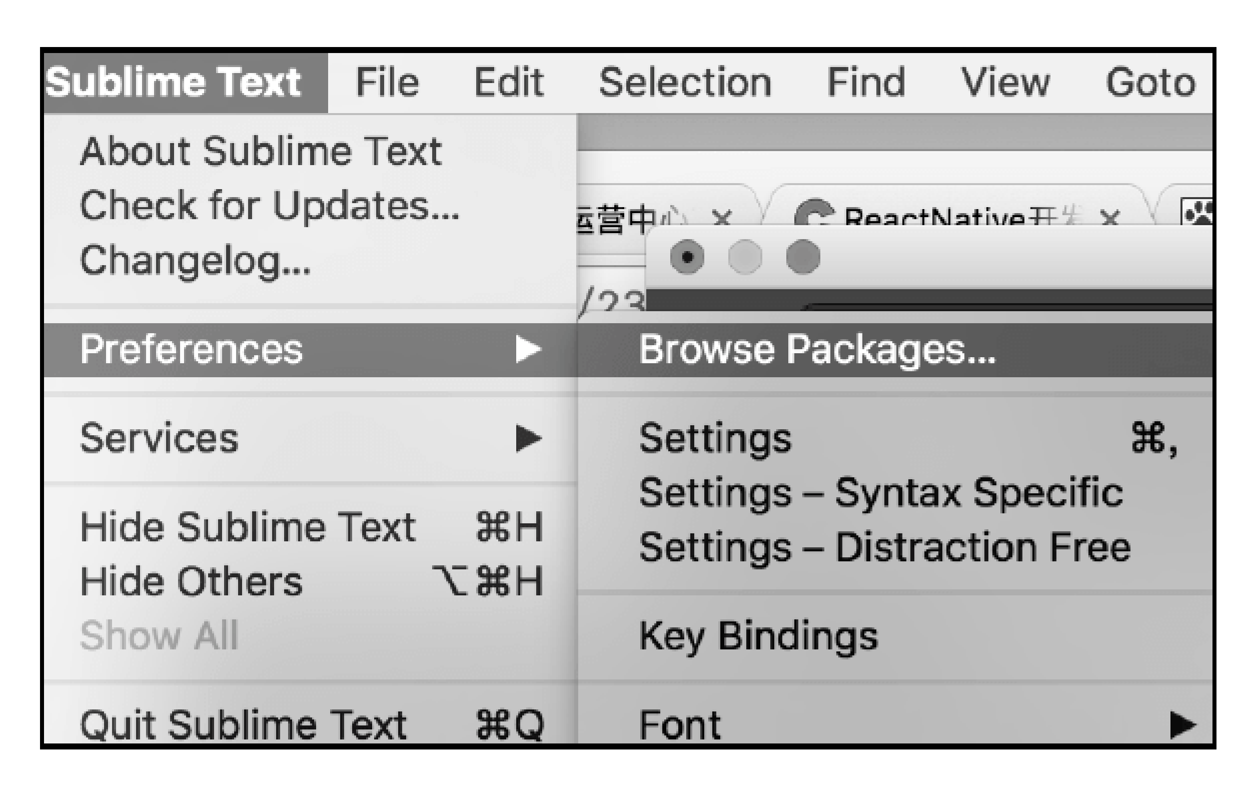


图 1-5　Sublime Text 的插件目录

将下载到的PackageControl安装文件放入Installed Packages文件夹中，如图1-6所示。



图 1-6　将 PackageControl 安装文件放入 Installed Packages 文件夹中

之后将Sublime Text完全关掉，重启Sublime Text，如果在Preferences中可以看到Package Control项目，说明已经成功安装，如图1-7所示。

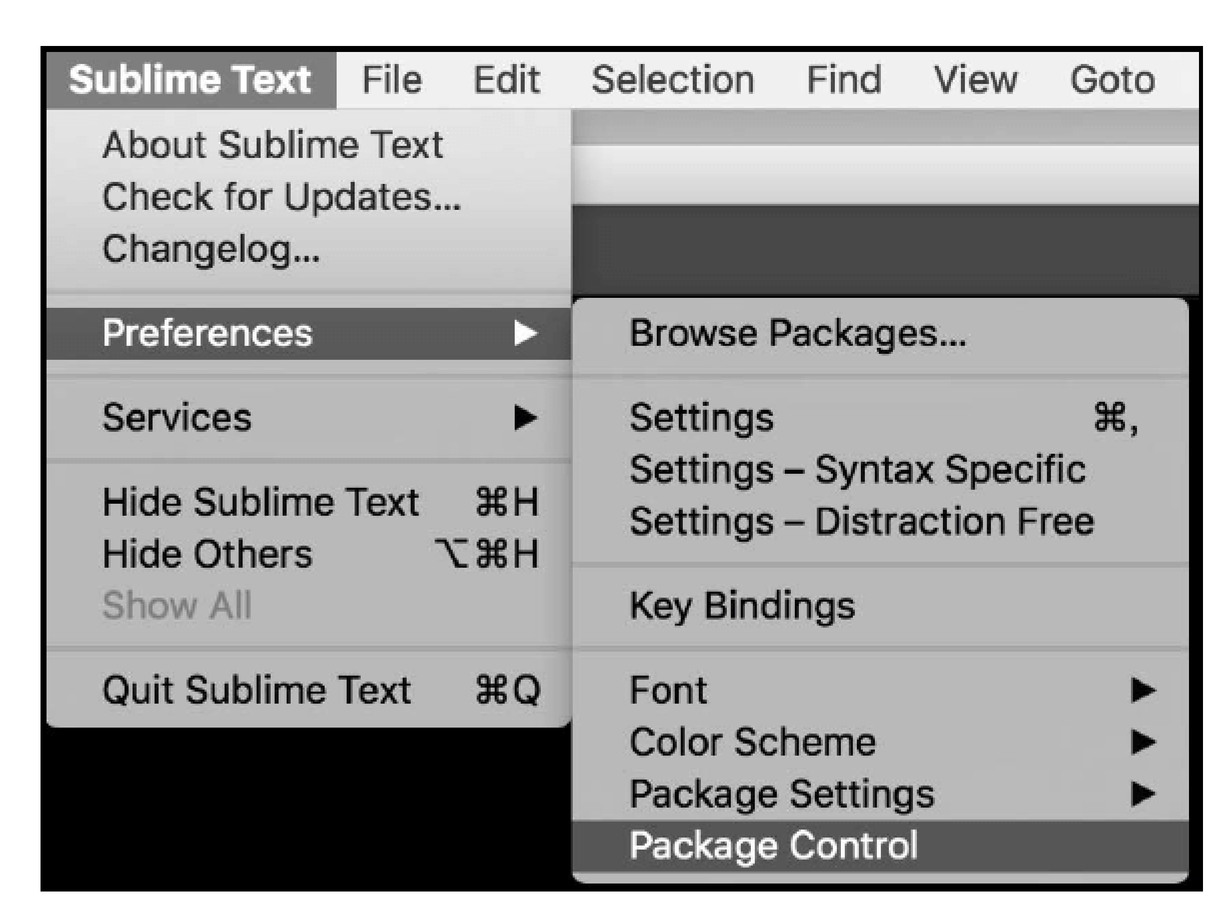


图 1-7　成功安装 PackageControl 工具

上面所提供的PackageControl安装包下载地址服务器在国外，在国内访问时常会出现波动，如果你无法从上面的地址下载PackageControl安装包，可以尝试如下地址：

http://zyhshao.github.io/file/Package%20Control.sublime-package

2. 使用 PackageControl 安装 JavaScript 代码智能提示插件 SublimeCodeIntel

对代码的智能提示是高级编辑工具必备的一项功能。SublimeCodeIntel是一个全功能的代码自动提示插件，支持众多流行的程序语言，例如JavaScript、HTML、CSS、Python、PHP等。

使用PackageControl可以十分方便地进行SublimeCodeIntel插件的安装。首先打开Sublime Text编辑工具，在Mac电脑中使用Command+Shift+P来打开PackageControl命令行，在其中输入package control便会自动检索出PackageControl工具所提供的所有命令，PackageControl工具中提供了许多易用的命令，如安装插件、查看已安装的插件列表、删除插件等，如图1-8所示。

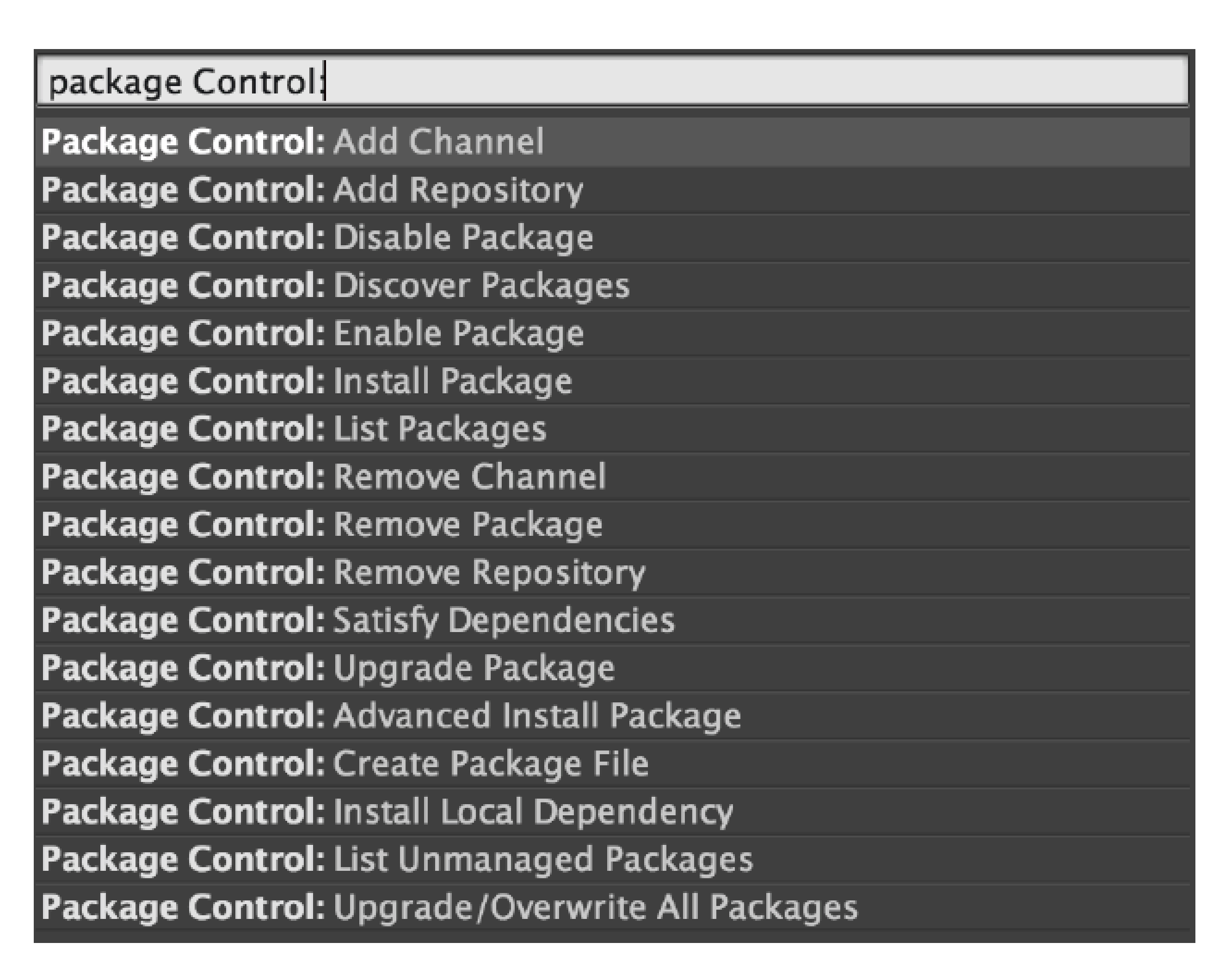


图 1-8　PackageControl 工具提供的命令

也可以通过如图1-9所示的方式打开PackageControl命令行：Sublime Text→Preference→Package Control。

在PackageControl命令行中输入Install Package并按Enter键就会进入安装插件的命令行，此时会自动显示插件列表，如图1-10所示。

在其中输入SublimeCodeIntel后按Enter键进行安装即可，如图1-11所示（安装需要数分钟时间）。如果安装成功，在Sublime Text→Preference→Package Setting中会看到SublimeCodeIntel项。

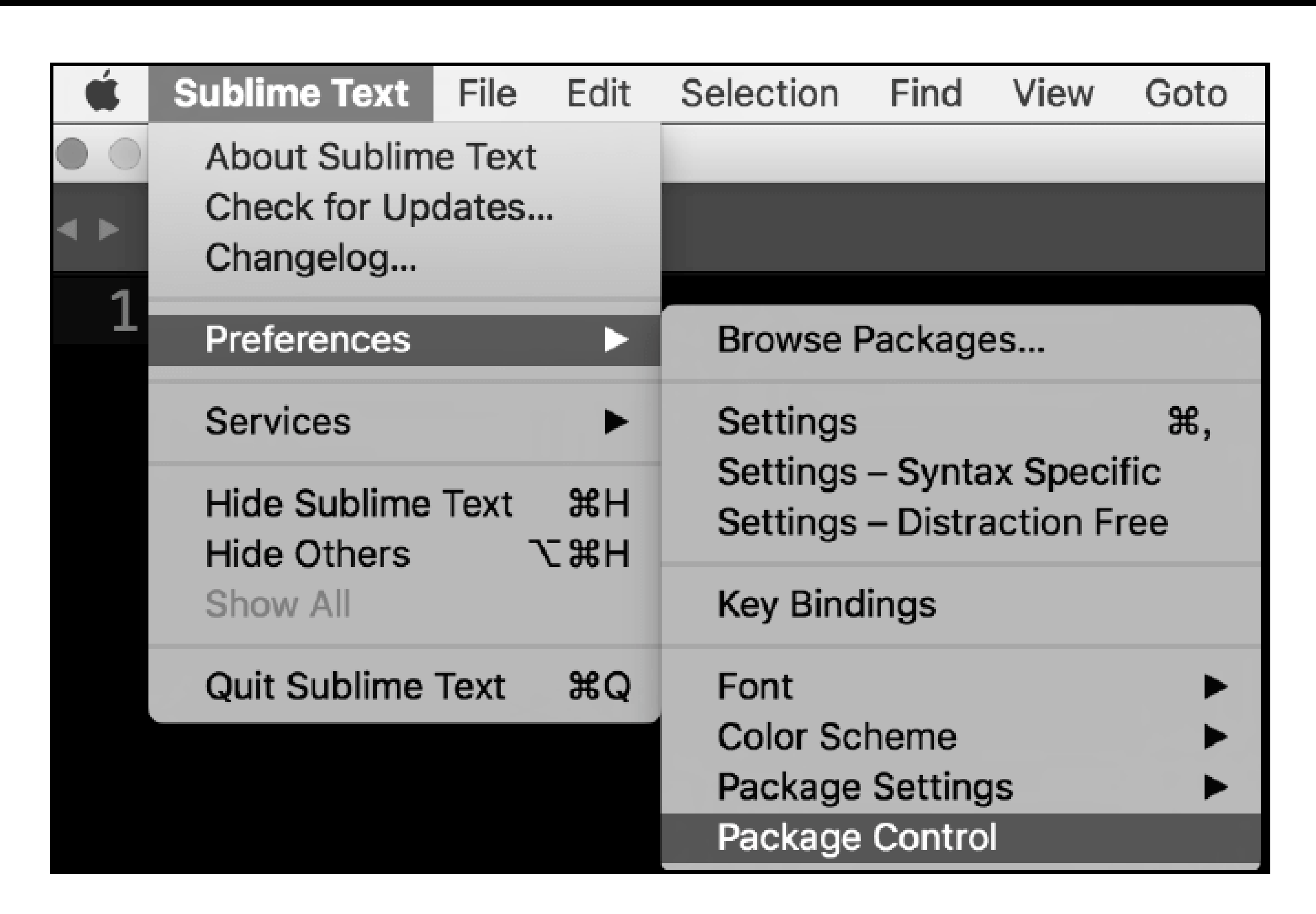


图 1-9　打开 PackageControl 命令行

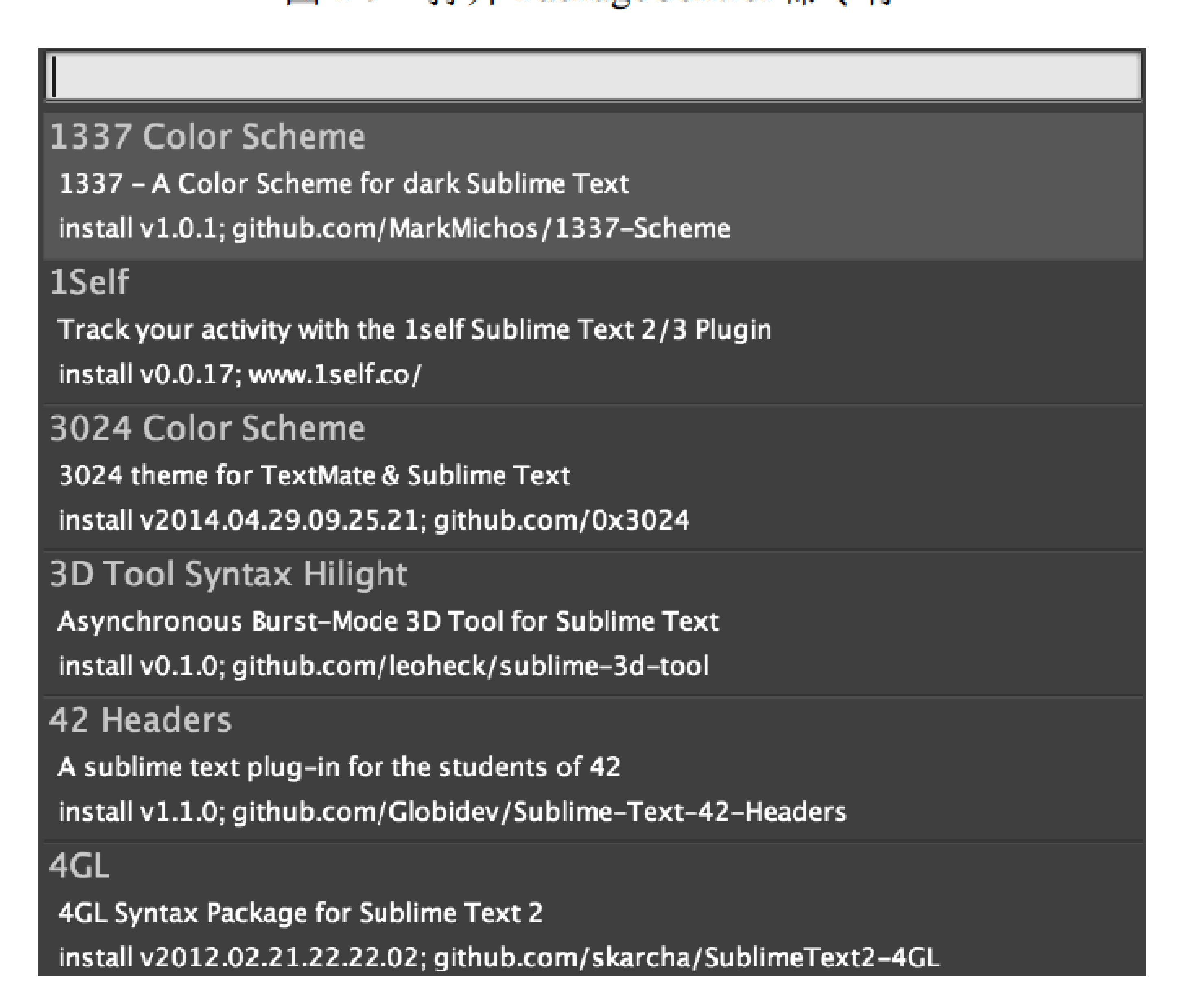


图 1-10　插件列表

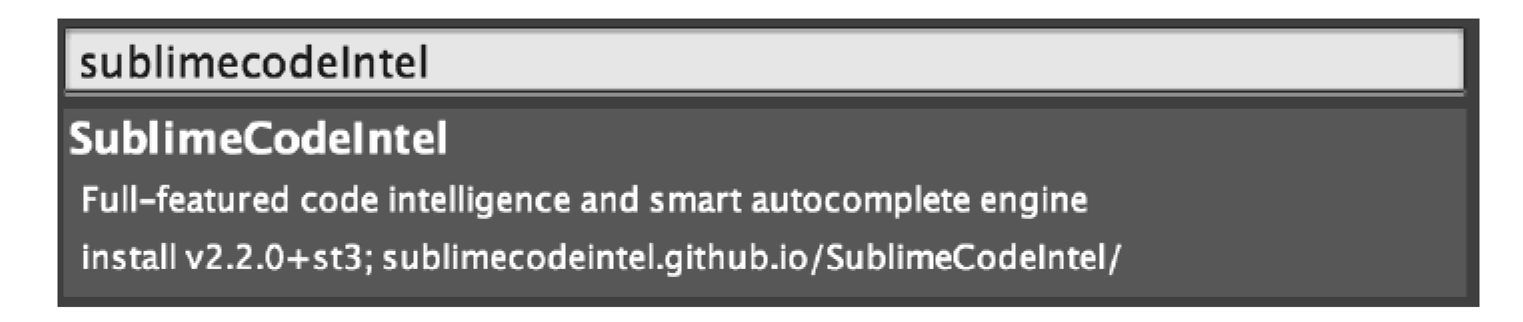


图 1-11　安装 SublimeCodeIntel 插件

在执行Install Package命令时，Sublime Text会从PackageControl官方网站拉取一个JSON文件，这个文件中包含所有Sublime Text的插件信息，大小在数兆左右。不幸的是，由于国

内网络的限制，这个文件的下载依然困难重重。笔者个人维护着一个此JSON文件的下载地址，无法成功拉取到文件的朋友可以尝试通过路径Sublime Text→Preferences→Package Settings→Package Control→Settings-User打开用户配置文件，如图1-12所示。

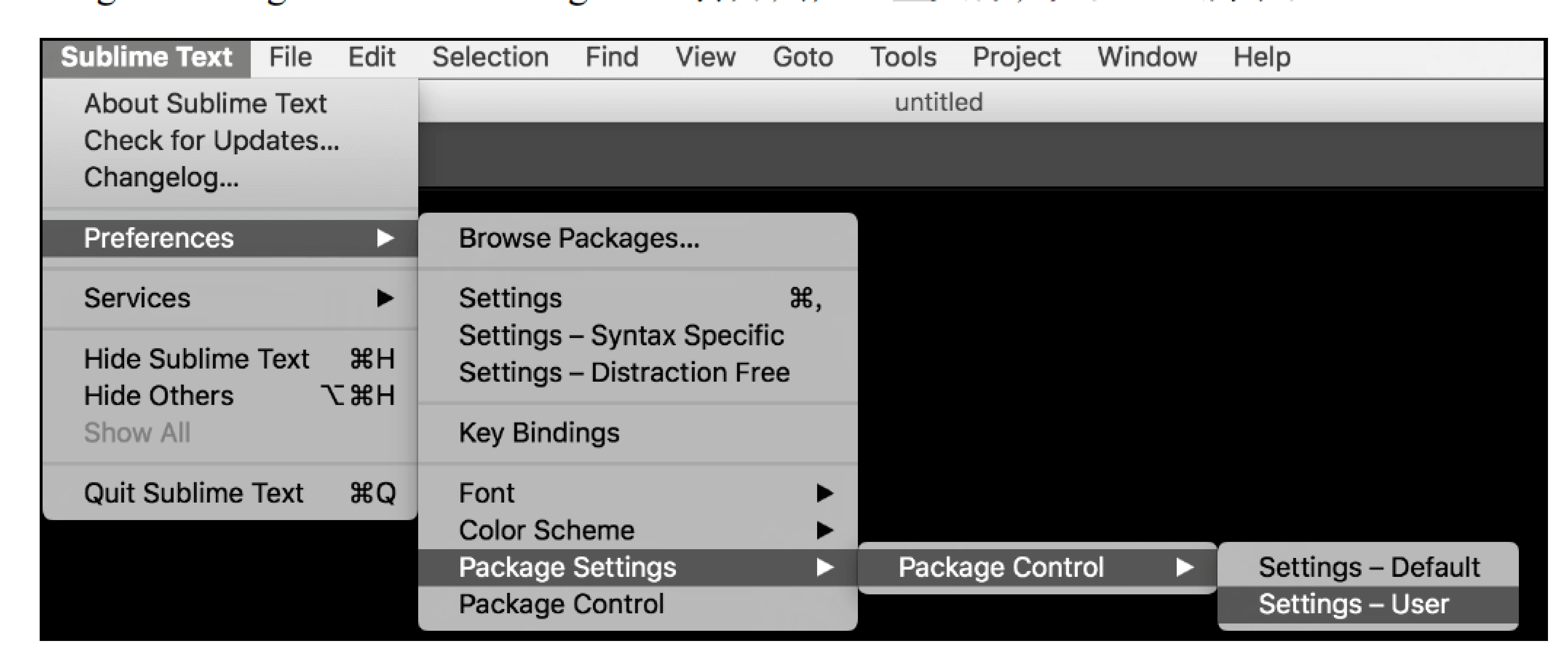


图 1-12　打开 Package Control 用户配置文件

在其中添加如下配置信息：

```
"channels": [
        "http://zyhshao.github.io/file/channel_v3.json"
    ],
```

添加完成后，配置文件看上去如图1-13所示。

```
{
    "bootstrapped": true,
    "installed_packages":
    [
        "Package Control"
    ],
    "channels": [
        "http://zyhshao.github.io/file/channel_v3.json"
    ],
}
```

图 1-13　进行用户配置文件设置

上面已经将插件列表JSON文件的拉取地址修改为http://zyhshao.github.io/file/channel_v3.json。

成功安装SublimeCodeIntel插件后，你可以尝试编写一些JavaScript代码，可以看到SublimeCodeIntel插件的智能提示十分强大，如图1-14所示。

```
Math.
abs (function)
acos (function)
atan (function)
ceil (function)
constructor (variable)
cos (function)
exp (function)
floor (function)
```

图 1-14　SublimeCodeIntel 的代码智能提示

3. 安装 JavaScript 代码格式化插件

缩进规范的代码会使我们在编写程序时赏心悦目，在PackageControl的插件列表中输入JsFormat，按Enter键进行此插件的安装，如图1-15所示。

安装成功后，同样可以在Sublime Text→Preferences→Package Settings中看到JsFormat项。

JsFormat插件的使用十分简单，选中要进行格式化的JavaScript代码，使用Control+Option（Alt）+F即可对其进行格式化。

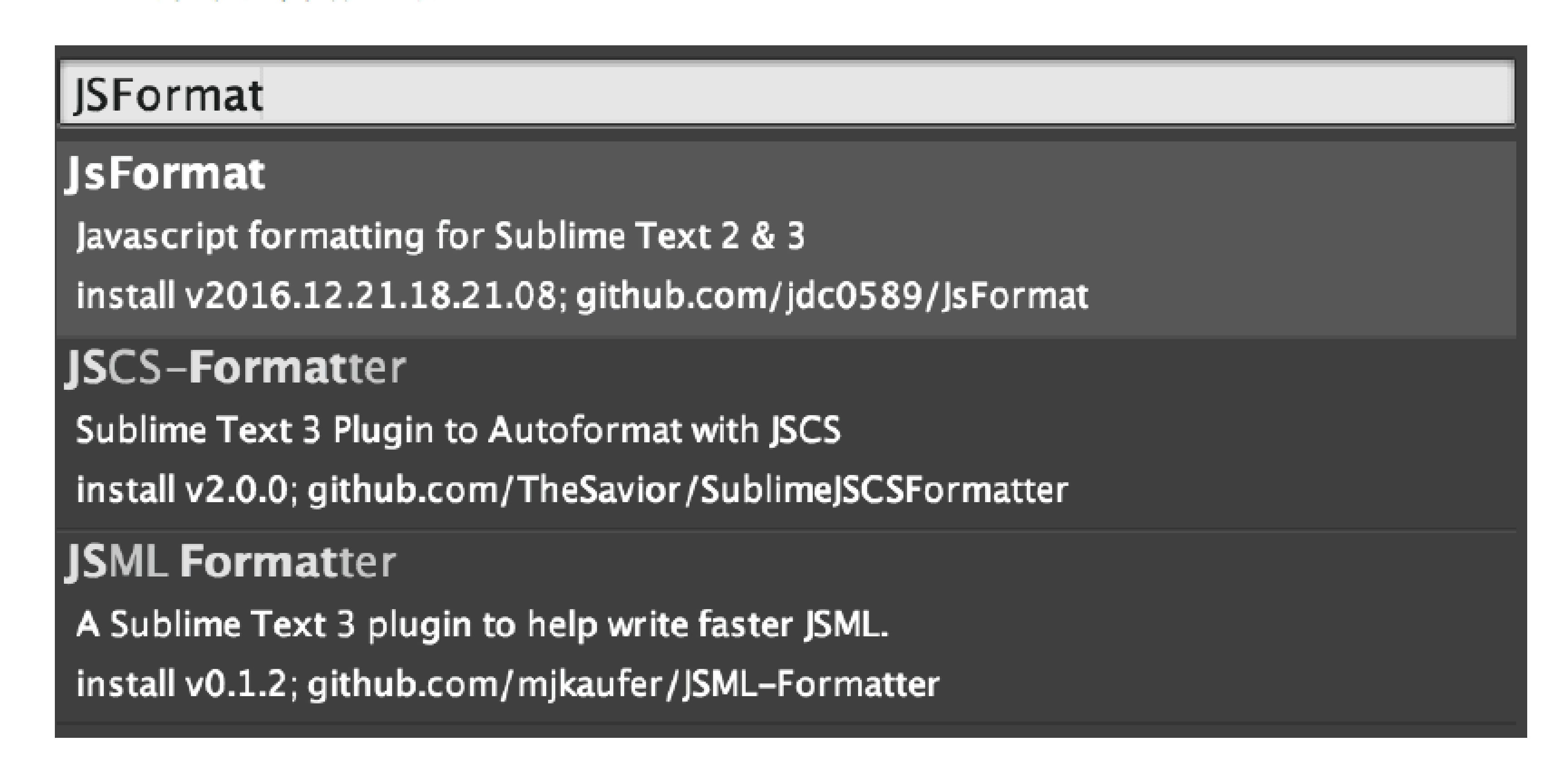


图 1-15　安装 JsFormat 插件

4. 在 Sublime Text 中运行 JavaScript 代码

前边我们做了很多工作，安装了一些易用的Sublime Text插件，这些工具可以帮助我们更加愉悦地进行JavaScript代码的编写。代码的编写是为了运行，以学习而言，能够实时地看到代码运行的结果也会大大提高学习效率，Sublime Text自带对Lua、Python、Ruby等语言的编译运行功能，但是并不支持JavaScript语言，我们做一些额外的配置，来使Sublime Text支持运行JavaScript代码。

Node.js是一款JavaScript运行时编译环境，首先需要在系统中安装Node.js环境，下载安装包地址：https://nodejs.org/en/。

安装完Node.js后，打开终端工具，在其中输入如下命令来查看Node.js的路径：

```
$ which node
```

打开Sublime Text工具，选择其中的Tools→Build System→New Build System，如图1-16所示。在新建的文件中写入如下信息：

```
{
    "cmd": ["/usr/local/bin/node", "$file"],
    "selector": "source.js"
}
```

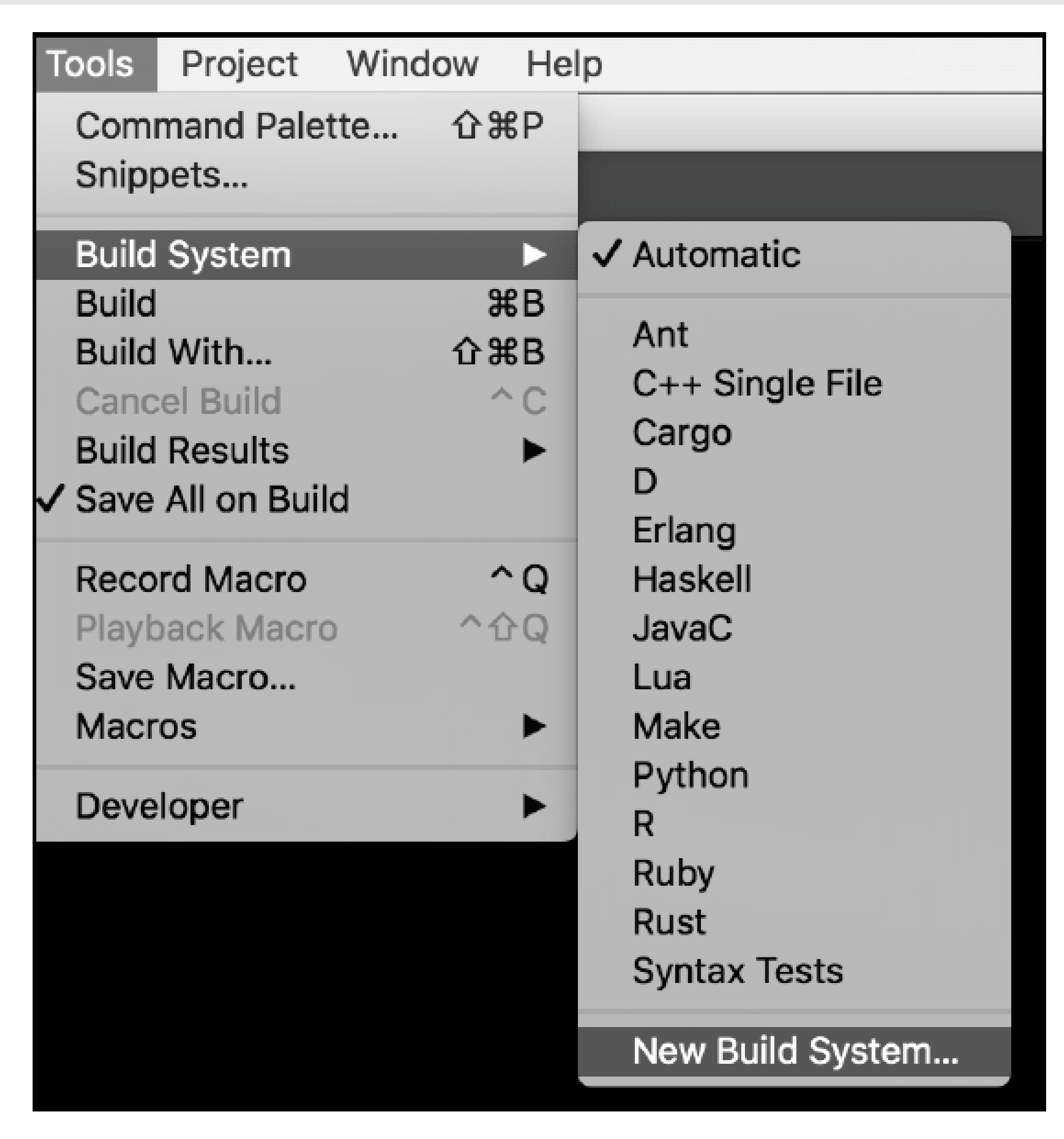


图 1-16　新建编译工具

需要注意，上面的/user/local/bin/node部分需要替换成你在终端中使用which node命令查找到的路径。之后将文件进行保存，命名为JavaScript.sublime-build即可。

新建一个Sublime Text文件，将其命名为text.js。将Tools→Build System中的编译工具选择新创建的JavaScript，编写一段JavaScript测试代码，使用Command+B进行代码的编译运行，可以看到，在Sublime Text控制台打印出了代码的执行结果与所耗时间，如图1-17所示。

到此，已经配置完成了一款十分快速且强大的JavaScript代码学习工具，后面我们将使用Sublime Text来一步步进入JavaScript的编程世界，一起玩起来吧！



图 1-17 在 Sublime Text 编辑器中进行 JavaScript 代码的执行

1-11 JavaScript 的语法特点

JavaScript是一种对大小写字母敏感的语言，也就是说，无论是变量名、函数名还是其他，都是区分大小写的。例如，下面的代码声明了两个完全不同的变量：

```
//大小写敏感
var name;
var NAME;
```

如果你熟悉一些编译型语言，例如C++、Java、Swift，那么你可能会固执地认为所有变量都要有强制的类型以确定其在内存中分配的空间大小。但是，学习JavaScript时，你需要忘记这条准则。JavaScript中的变量是动态弱类型的，你可以将一个变量先赋值为字符串类型的值，再将其修改为数值类型的值。总之，JavaScript中的变量没有特定的类型，你可以将其赋值为任意类型的值。示例如下：

```
//动态类型
var dy = "string"       //字符串
dy = 1                  //数字
dy = true               //布尔
```

虽然JavaScript允许对一个变量进行多种类型值的赋值，但是在开发中，笔者还是强烈建议不要这样做，规范与固定意义的变量会使你的项目看起来赏心悦目。

在JavaScript中，每行结尾的分号可有可无，这一点十分类似Swift语言，但是需要注意，如果你在同一行中写了多条语句，需要使用分号进行语句的分隔。良好的代码风格习惯是一行内只写一条语句，并且加上分号，这样做不仅方便进行代码的压缩，而且在调试时可以很好地暴露出现问题代码所在的位置。示例如下：

```
//关于分号
var value1 = 1;
var value2 = 2; console.log(value2);
var value3 = 3    //行末尾可以省略分号
```

在JavaScript中，可以使用反斜杠进行字符串的折行编写，有些时候这样做可以使你的代码看起来更加漂亮，示例如下：

```
//使用反斜杠进行字符串的折行
var value4 = "\
Welcome to JavaScript \
My Good Friend!";
console.log(value4);
```

任何编程语言都会提供注释的能力，一个优秀的开发者不仅会写代码，更需要会写注释。JavaScript中有两种方式进行注释的编写，其完全遵照了C语言的注释风格。使用双斜杠进行单行注释，使用双斜杠中间嵌入两个星号来进行多行注释，示例如下：

```
//这里是单行注释
/*
这里是多行注释
*/
```

1-12　编程练习

练习1：在Chrome浏览器的调试模式下，使用alert函数在当前网页上弹出一个警告框。

解析：

在Chrome浏览器中使用快捷键Command+Option+J可以快速打开调试模式，在Console区键入如下代码后按回车键：

```
alert("你好，Google");
```

弹出窗口效果，如图1-18所示。



图 1-18　在网页弹出警告框

练习2： 使用JavaScript表达式描述语句：3加5的和与6的乘积除以12。

解析：

```
var result = (3+5)*6/12;
```

练习3： 使用JavaScript实现简单逻辑：如果小明的年龄大于12岁，小明就可以参加夏令营，否则不能参加夏令营，小明的真实年龄是10岁。

解析：

```
var xiaomingAge = 10;
if(xiaomingAge > 12){
        console.log("可以参加夏令营");
}else{
        console.log("不能参加夏令营");
}
```

练习4： 试着使用JavaScript编写求10的阶乘的运算。

解析：

```
let result = 1;
for(let i = 1;i<=10;i++){
        result = result*i;
}
console.log(result);            //3628800
```

练习5：将例4中的计算过程封装成函数。

解析：

```
function func(param){
    let result = 1;
    for(let i = 1;i<=10;i++){
        result = result*i;
    }
    return result;
}
console.log(func(10));        //3628800
```

练习6：对象是JavaScript描述复杂数据类型的方式，试着用对象来描述一辆商品汽车。

要求：奔驰牌汽车，价格是2 500 000元，颜色是红色的，有自动驾驶和人工驾驶两种行驶方式。

解析：

```
var car = {
    brand:"奔驰",
    price:"2500000",
    color:"red",
    go:function(isAuto){
        if (isAuto) {
            console.log("正在自动驾驶");
        }else{
            console.log("正在人工驾驶");
        }
    }
}
```

练习7：使用数组存放班级10名学生的期末考试成绩：67、76、87、56、98、98、67、89、78、67。

解析：

```
var record = [67，76，87，56，98，98，67，89，78，67];
```

第 2 章

ECMAScript 的语法世界

ECMAScript是JavaScript的核心语法标准，它是JavaScript中最重要的组成部分。本章将通过一系列范例来帮助你窥探ECMAScript的语法世界。对于一门高级编程语言，在学习时你只需要把握两条主线：面向过程与面向对象。在学习面向过程时，要注意语言的变量、函数、运算、流程、分支、循环、跳转等关键点；在学习面向对象时，则要注意语言的对象、类、属性、方法、继承、扩展等关键点。JavaScript语言有高效的面向过程的特点，又有强大的基于原型的面向对象的能力。现在，我们就一起开始探索这门神奇的语言吧。

2-1　理解变量

变量一词来源于数学，其代表函数中能够发生改变的量值。在计算机语言中，用于存储计算结构或表示值的抽象概念。需要注意，变量有可能是可变的，也有可能是不可变的，变量具体的意义由不同的编程语言所定义。在JavaScript中，使用var（var是variable单词的缩写）、let和const关键字来进行变量的声明。

如果你看过一些JavaScript程序，可能会发现其中充斥着大量var关键字。确实如此，let和const关键字是ECMAScript 6之后引入的新特性，老版本的ECMAScript中只有var一个关键

字来进行变量的声明和定义。关于声明与定义，最大的区别是：声明只是在程序中预定了一个变量名称，不需要进行存储空间的建立；定义则是对变量进行赋值，需要建立存储空间。示例代码如下：

```
//变量的声明与定义
var name;                //进行变量的声明
var age = 25;            //进行变量的定义
```

你也可以在同一语句中进行多个变量的声明或定义，例如：

```
//进行多个变量的声明
let a,b,c,d,e=3;
console.log(a,b,c,d,e);        //undefined undefined undefined undefined 3
```

let和const关键字在声明和定义变量时，语法和var关键字完全一致。不同的是，let声明的变量会受作用域的影响，const定义的变量不能够被修改，也可以将其理解为“常量”。

2-2　变量的命名

在对变量进行命名时，需牢记下面两条规则：

（1）变量名的第1个字符必须是字母、下画线或者美元符号。
（2）除了第1个字符之外，其余字符可以是下画线、美元符号或者任意数字与字母。

下面这些变量名都是合法的：

```
var _myName_,MyName,$name,_3name,n3;
```

下面这些变量名都是非法的：

```
//不合法的变量名
var 3l;
var %2;
```

虽然JavaScript对变量的命名比较自由，但并不意味着开发者在命名变量时可以随心所欲。正确地对变量命名应该能够做到见形知意，并且从外观上看起来不突兀，很自然。比较著名的变量命名方法有如下几种：

1.Camel（驼峰）命名法

Camel命名法是指变量的首字母小写，接下来的每个单词的首字母大写，示例如下：

```
//驼峰命名法
var myName;
```

2. Pascal 命名法

Pascal命名法是指变量的首字母进行大写，其后每个单词的首字母也进行大写，Pascal命名法有时也被称为大驼峰命名法，示例如下：

```
//Pascal 命名法
var MyName;
```

3. 匈牙利类型命名法

Camel与Pascal命名法只针对变量的意义进行解释，匈牙利类型命名法中还加入了变量的类型，其规则是在Pascal命名法的基础上，在变量名的最前面加上变量类型的标识。例如数字型变量添加i标识、字符串变量添加s标识，示例如下：

```
//匈牙利类型命名法
var iAge = 25;
var sName = 'jaki';
```

表2-1列出了常用类型对应的标识。

表2-1　常用类型对应的标识

类　型	标　识
数组	a
布尔	b
浮点数字	f
整数	i
函数	fn
对象	o
正则表达式	re
字符串	s
任意类型	v

另外，对于一些大小写不敏感的编程语言，也常常采用下画线命名法。

4. 下画线命名法

单词与单词之间使用下画线进行分割，示例如下：

```
//下画线命名法
var my_name;
```

作用域与作用域链

作用域对于一门编程语言至关重要，在许多编程语言中，都以大括号进行作用域的划分。例如，C语言中的for循环体、while循环体、if分支块等构成一个作用域，在其中定义的变量只在作用域内有效，出了作用域则不能被访问到。在ECMAScript中，除了块级作用域外（与let相关），作用域是以函数来进行区分的，初学者往往会在这里产生迷惑。

作用域控制着变量的可见性与生命周期。在进行软件设计时，开发者应该遵循最小暴露原则，将一些不必要的变量和函数隐藏起来。ECMAScript中的作用域可以笼统地划分为两类：全局作用域与局部作用域。全局作用域中的变量和函数在代码中的任何地方都可以访问到（最外层函数和定义的变量），例如：

```
//全局作用域
function globalFunc(){
    console.log("globalFunc");
}
var name = 'Jaki';
var age = 25;
```

上面代码中的函数globalFunc、变量name和age都在全局作用域内。

```
//局部作用域
function subBlock(){
    var subject = 'JavaScript';
    function show(param){
        console.log("subBlock "+param);
    }
    show(subject);
}
// console.log(subject);     //程序会抛出异常
// show("s");                //程序会抛出异常
subBlock();                  //subBlock JavaScript
```

上面代码中的subBlock函数创建了一个局部作用域，其中的变量subject和函数show都只能在其作用域内进行访问。

另外，在进行变量访问时，ECMAScript会遵循作用域链的方式从内到外逐层访问，如果在内层作用域中可以访问到变量，就会停止寻找，示例如下：

```
function func(){
    console.log(name);          //访问到全局作用域的 name
}
function func2(){
    var name = '珲少';
    console.log(name);          //访问到局部作用域的 name
}
func2();        //珲少
func();         //Jaki
```

由此可知，在ECMAScript中，如果你频繁访问一个全局作用域中的变量，将是十分影响性能的。

2-3　变量提升

ECMAScript中有一个十分奇怪的语法规则：变量提升。如果你使用了一个从未声明过的变量，程序运行会直接抛出异常。但是如果代码中有过对此变量的声明，无论在声明前使用还是在声明后使用，程序都不会抛出异常，例如：

```
//变量提升
console.log(name);          //undefined
var name = "Jaki";
console.log(name);          //Jaki
```

还有一点，如果你直接对一个未经声明的变量进行赋值，就相当于直接在全局作用域中创建了这样一个变量，这是一种十分危险的做法，会造成无意的变量泄露，例如：

```
//变量泄露
function func(){
    age = 24;       //泄露为全局变量
}
func();
console.log(age);       //24
```

造成这种语法特性的原因是：JavaScript解释器在对代码进行扫描时，会将全局作用域中声明的变量和函数先定义为全局符号，运行到具体声明处才进行赋值。这种语法特性多多少少会对开发者造成一些误解，在许多流行的编程语言中，变量在声明之前是不能使用的。如果说上面的示例代码你仍然觉得没有什么好紧张的，那么下面的代码就能说明问题了：

```
//变量提升
console.log(name);          //undefined
if (false) {
    var name = "Jaki";
}
for (var i = 0 ; i < 3; i++){
    var sum = i;
}
console.log(i);             //3
console.log(sum);           //2
```

上面的示例代码中，首先if条件判断语句值为假，if代码块永远不会执行到，但是从打印结果可以看出，第一句打印并未抛出异常，name变量还是被声明了。而后面的循环结构中，我们在for循环条件结构中声明了一个i变量，在循环体内声明了sum变量，可是当循环结束后，i变量依然存在，变成了全局变量。sum变量同样也变成了全局变量，这种所谓的“变量提升”会消耗一部分无用内存，并对之后的代码编写产生额外的风险。在ES6标准中，新引入块级作用域可以完美地解决这些问题，下一示例中再来研究let与块级作用域。

2-4　块级作用域

ES6标准中的块级作用域实际上是由let与const关键字决定的。在ES6标准中新引入了let和const关键字，其用法与var关键字十分类似，都是进行变量的声明。不同的是，var声明的变量会存在变量泄露和变量提升，从而成为全局变量的问题。而let和const声明的变量则只在其所在的代码块中有效。所谓代码块，即由大括号包裹起来的区域，其可以是一些常用的语法结构，如分支结构、循环结构等，也可以是开发者自行创建的区域。示例如下：

```
//块级作用域
{
    var a = 10;
    let b = 10;
    console.log(b);         //10
}
console.log(a);             //10
console.log(b);             //程序抛出异常
```

上面的代码当程序运行到console.log(b)时会抛出异常。也就是说，使用let命令声明的变量，一旦脱离其所在的代码块，这个变量就不能再被使用。这种局部变量十分适合用于for循环结构，例如：

```
for(let i=0;i<3;i++){
}
console.log(i);          //程序抛出异常
```

let命令还有一个规则，其不存在变量提升，即在变量声明之前，此变量是不可使用的，而不是undefined，例如：

```
console.log(a);          //程序抛出异常
let a = 10;
```

使用let声明的变量也不可以进行重复声明，如下的写法也会抛出异常：

```
let a = 10;
let a = 11;              //程序抛出异常
```

关于块级作用域还有一个特点需要特别注意，如果在块级作用域内使用let或const声明了某个变量，那么此作用域会形成对此变量的屏蔽。也就是说，即便外层作用域中也有同名的变量，也会被屏蔽掉，这种语法特性被称为暂时性死区，示例如下：

```
let tmp = 10;
{
    console.log(tmp);          //程序抛出异常
    let tmp = 11;
    console.log(tmp);          //11
}
```

上面的示例代码说明，在块级作用域中使用let声明了变量，那么在声明之前，这个变量都不能使用（尽管全局中也定义了同名的变量）。

我们回过头再来理解一下块级作用域。在ES5标准中是没有块级作用域这个概念的，我们在编写代码时，很容易产生内层变量覆盖外层变量和局部变量泄露为全局变量的问题。块级作用域使作用域内的变量不受外界影响，同时也不会影响外界，提高了代码的安全性。同时，块级作用域也是可以嵌套的，外层作用域无法读取内层作用域的变量，示例如下：

```
//将打印
/*
Hello World
Wa
New
*/
{
    let a = "New";
    {
```

```
            let a = "Wa";
            {
                let a = "Hello World";
                console.log(a);
            }
            console.log(a);
        }
        console.log(a);
    }
```

对于函数的声明，在ES6中也是遵守块级作用域规则的，在作用域内声明的函数只能在作用域内使用，例如：

```
    {
        let func = function(){
            console.log("function");
        }
        func();              //function
    }
    func();              //程序抛出异常
```

2-5　ECMAScript 中的数据类型

变量是用来存储特定意义的值。在JavaScript中，变量可以存储两种类型的值：原始值和引用值。原始值和引用值的区别在于存储的位置与访问的方式不同。原始值是存储在栈中的简单数据，引用值是存储在堆中的对象数据。也就是说，当你通过变量名访问原始值时，会直接访问到其存储在栈中的数据；而通过变量名访问引用值时，会首先获取存储在栈中的对象地址，根据地址再向堆中查找真正的对象数据。ECMAScript中定义的原始类型有5种，分别为Undefined（未定义类型）、Null（空对象类型）、Boolean（布尔类型）、Number（数字类型）、String（字符串）类型。图2-1描述了原始值与引用值的差异。

原始值所占的内存大小一般是固定不变的，将其存储到栈中可以更快地进行数据访问。而引用值所占的内存通常较大并且不固定，但其地址所占的内存大小是固定不变的，将其地址存入栈中不会影响性能。在5种原始类型中，String类型十分特殊，因为其大小也是不固定的。在Java、Objective-C等语言中，字符串通常会被定义为引用类型，但JavaScript中依然将其作为一种原始类型。

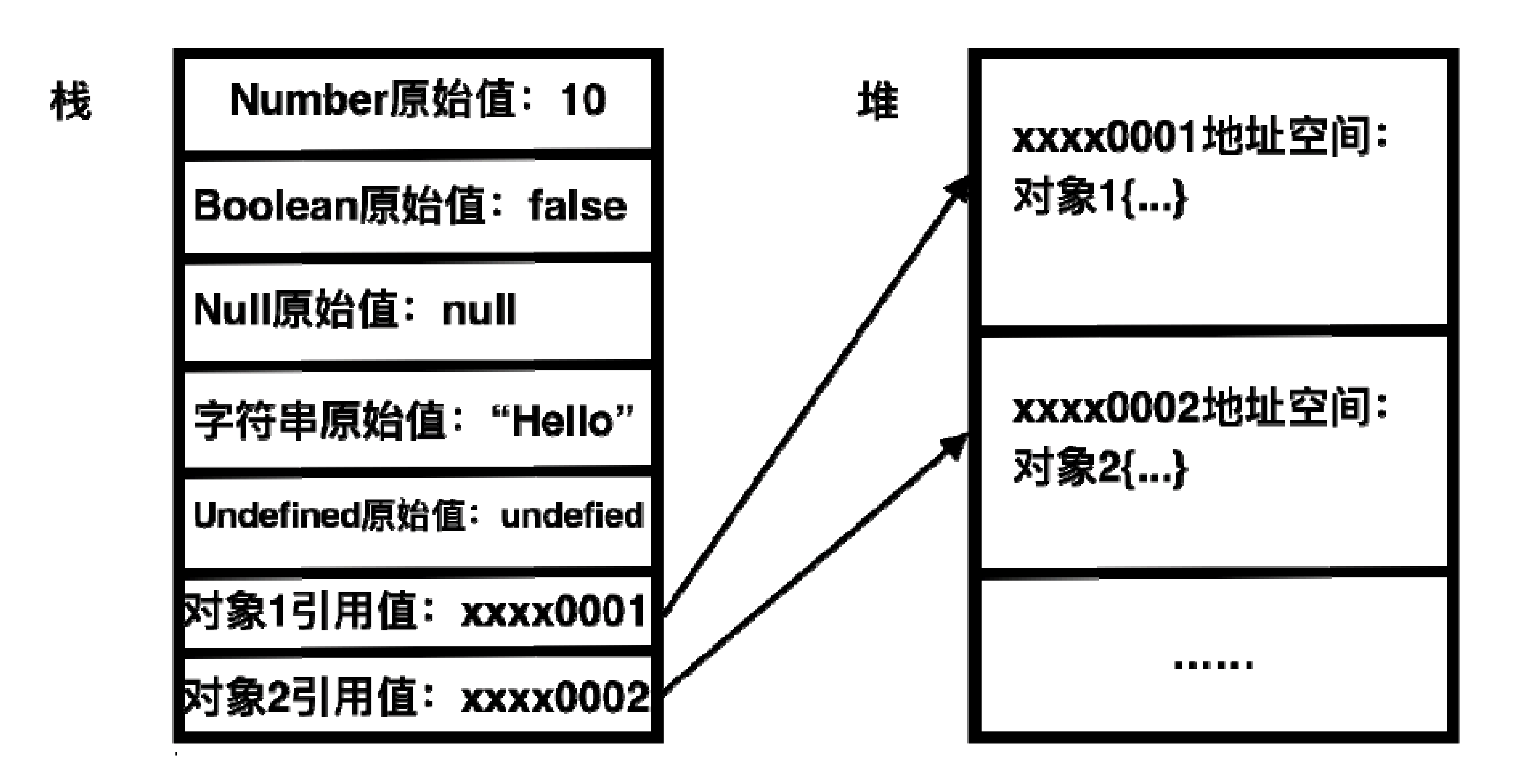


图 2-1　原始值与引用值图示

2-6　再看 const 关键字

我们前面提到，const关键字用来声明的变量不可修改。在许多编程语言中，除了有变量的概念，还有常量的概念。常量就是值不能改变的量，在ES6标准中，使用const关键字来进行常量的声明。修改常量的值会使程序抛出异常，示例如下：

```
const PI = 3.14;
PI = 3;            //抛出异常
```

需要注意，在使用const声明变量时，要同时为其进行赋值，一旦const变量被定义，后面就不能够再对它进行修改。const关键字声明的变量和let关键字声明的变量享有同样的作用域规则，这里不再赘述。

const声明的常量有一点需要额外注意，即const实际保证的是常量空间存储的数据不可被修改，而常量所对应的值有时是可以修改的。例如，常量对应的是一个对象，你可以修改此对象的属性和方法，但不可以直接将此常量指向的对象修改掉，示例如下：

```
const teacher = {
    name:"Jaki",
    age:25
};
//对对象进行修改没问题
teacher.name = "Lucy";
teacher.age = 24;
```

```
//直接修改常量的指向则会报错
teacher = {
        name:"Lucy",
        age:24
};
```

2-7　Undefined 与 Null

前面讲过，JavaScript中定义的原始类型有5种：Undefined、Null、Boolean、Number、String。切记，只有这5种原始类型。原始类型数据是直接使用字面值来创建的。其中，Undefined和Null类型比较特殊。

Undefined类型只有一个值，即undefined。其意义也如所描述的那样，即未被定义的。例如，一个变量只是被声明，其值就是undefined，其类型就是Undefined类型。JavaScript中的typeof关键字可以用来获取变量或值的类型，示例如下：

```
//Undefined 类型
var unKnown;
console.log(typeof unKnown);     //将打印 undefined
```

需要注意，仅仅被声明但未被赋值的变量是未定义的，实际上从没有声明过的变量也是未定义的，示例如下：

```
//未声明过的变量也是未定义的
console.log(typeof unKnown2);   //将打印 undefined
```

typeof是一个特殊的运算符，但如果将未声明过的变量用于其他运算符，将会产生运行错误。

执行一个没有返回值的函数后，也会返回undefined值，示例如下：

```
//1 个无返回值的函数
function func(){
        console.log("func");
}
var v1 = func();          //将打印 func
console.log(v1);          //将打印 undefined
```

可能你现在对函数还不太理解，不用担心，后面我们会专门学习函数的相关内容。

Null类型是ECMAScript中另一种只有一个值的类型，其字面值为null。Null类型的定义唯一的用途是作为空对象的占位。现在，你可能对对象也不太理解，在介绍原始值与引用值

时提到过对象，对象是一种复杂数据类型，对象变量中实际存储的是对象的引用地址。ECMAScript中的对象不属于原始类型，原则上它们之间并不会产生强关系，然而在实际开发中，开发者往往需要一种约定的值来表示空对象，即要有一个约定的值来描述一个无用的地址，这个值就是null。开发者在使用对象前，发现变量中存储的引用地址为null时，就知道此对象还没有被初始化，或者此对象已经不存在了。

如果在JavaScript中使用typeof来对null值进行类型检查，你会惊奇地发现其返回的类型是object，这或许是JavaScript前期实现上的一个错误，但是其也恰恰与null是空对象的占位这一概念完全契合。因此，在ECMAScript标准中沿用了这一定义。示例如下：

```
//Null 类型
var obj = null;
console.log(typeof obj);            //将打印 object
```

现实世界中，很多设计可能都不是最正确的，却无疑是最合适的。一个很有趣的例子来自键盘设计。客观地说，目前主流键盘布局并不科学，这种“QWERTY”布局的键盘设计之初是供打字机用的，而“ABCDEF”排序的键盘在打字速度过快时，往往会产生卡顿问题。为了解决这个问题，克里斯托夫·拉森·肖尔斯刻意将高频字符放置在相反的方向，以最大限度地放慢敲键速度。也就是说，现代键盘的布局设计是为了降低打字速度。另一种更科学的键盘布局方式为DVORAK布局，如果在互联网上搜索DVORAK关键字，你会搜出很多理由阐述这种布局的好处，然而其依旧无法成为主流，人们的习惯根深蒂固且不会轻易尝试去改变。

2-8 关于 Boolean 类型

Boolean定义的原始值有两个，分别为true和false。通俗地讲，Boolean值就是用来描述事物的真与假，是与非的概念，其在逻辑运算中有着很广泛的应用，一个简单的条件结构示例如下：

```
var number = 10;
var result = number>10;
if (result) {
    console.log("大于 10");
}else{
    console.log("不大于 10");
}
//将打印不大于 10
```

上面代码中的number>10其实是一种比较运算，其运算的结果就是Boolean值。

2-9　关于 Number 类型

Number类型用来描述数字，和其他编程语言不同的是：JavaScript中的Number类型既可以描述整数值，也可以描述浮点值。示例如下：

```
//Number 类型
var num1 = 100                    //整数值
var num2 = 3.14                   //浮点值
console.log(typeof num1);         //将打印 number
console.log(typeof num2);         //将打印 number
```

在数值前添加前缀可以将其描述为八进制或十六进制的数值。八进制需要将0作为前缀，十六进制需要将0x作为前缀，示例如下：

```
//八进制
var num3 = 011;              //对应十进制 9
//十六进制
var num4 = 0x11;             //对应十进制 17
```

需要注意，很多编程语言并不介意数值量前面是否添加前缀0，JavaScript语言对这一点要求十分严格，多余的0会改变数值的进制方式，造成不可控的错误。

对于非常大或非常小的数值，JavaScript中也可以使用科学计数法进行描述，使用字母e来描述10的e次方，示例如下：

```
//科学计数法
var num5 = 1.01e3;                //对应 1010
var num6 = 1111000e-6;            //对应 1.111
```

JavaScript中还定义了一些特殊的数值，Number.MAX_VALUE和Number.MIN_VALUE分别用来表示Number类型所能表示的最大值与最小值，示例如下：

```
//Number 最大可以表示的值
console.log(Number.MAX_VALUE);          //打印 1.7976931348623157e+308
//Number 最小可以表示的值
console.log(Number.MIN_VALUE);          //打印 5e-324
```

当计算值超出了Number类型所能表示的极限时，会被认作无穷。JavaScript中也专门定义了特殊的Number值来表示无穷，其中Number.POSITIVE_INFINITY表示正无穷大，

Number.NEGATIVE_INFINITY表示负无穷大，它们的值分别为Infinity与-Infinity，示例如下：

```
//正无穷
console.log(Number.POSITIVE_INFINITY);          //将打印 Infinity
//负无穷
console.log(Number.NEGATIVE_INFINITY);          //将打印-Infinity
```

JavaScript中定义的最后一个比较特殊的Number值为NaN，为Not a Number的缩写，表示不是一个数字。这个值在字符串向数字转换失败时会被返回，示例如下：

```
//NaN 值
var num7 = Number("w");
console.log(num7);          //将打印 NaN
```

需要注意，NaN这个值十分特殊，其不可以进行计算也不可以进行比较，并且与其自身也不相等，例如如下的比较将会返回false：

```
console.log(NaN == NaN);          //将打印 false
```

如果要判断一个变量的值是否是NaN，需要使用如下方法：

```
console.log(isNaN(num7));          //将返回 true
```

2-10　关于 String 类型

String类型是ECMAScript中唯一没有固定大小的原始类型，用来存储多个Unicode字符。在C、Java等语言中，字符和字符串是两种不同的类型，字符使用单引号包裹，字符串则使用双引号包裹。在ECMAScript中删去了字符的概念，字符串可以使用单引号包裹，也可以使用双引号包裹，但是如果要在字符串中嵌套字符串，单双引号必须交替使用。示例如下：

```
//String 类型
var str1 = "Hello";
var str2 = 'World';
var str3 = "Hello 'World'";
```

和C、Swift、Java、Perl等语言类似，JavaScript中也定义了一些转义字符，如表2-2所示。

表2-2　JavaScript中的转义字符

转义字符	含　义
\n	换行
\t	制表符
\b	空格
\r	回车
\f	换页符
\\	反斜杠
\'	单引号
\"	双引号
\0nnn	使用八进制代码表示字符
\xnn	使用十六进制代码表示字符
\unnnn	使用十六进制 Unicode 码表示字符

某些编程语言会定义专门的函数来拼接处理字符串。当然在ECMAScript中，String对象里也定义了许多操作字符串的方法。对于字符串拼接，更简单的方法是直接使用加法运算符，示例如下：

```
//字符串的拼接
var str4 = "hello"+" "+"world";
console.log(str4);        //hello world
```

2-11　对象简介

ECMAScript中的引用类型实际上指的就是对象。对象是一组功能行为互补的数据集合，其可以用来模拟实现生活中的事物。举一个简单的例子，如果要开发一款教学系统，这个系统中需要包含老师和学生两类成员。其中，老师就是一种对象，学生也是一种对象。老师对象中可能会包含姓名、教师编号、专业、所带班级等，学生对象中可能会包含姓名、年龄、所学课程、所在班级等。当然，除了这些描述对象属性的数据外，对象中还需要包含一些行为，例如老师要进行教学、学生要学习考试等。在ECMAScript中，Object类型就是这样一种引用类型，其创建出来的实例被称为对象。

对象的创建有两种方式，第一种，可以直接通过Object构造方法来新建对象。以教师对象为例，代码如下：

```
//创建教师对象
var teacher = new Object();
//为教师对象添加一些属性
teacher.name = '珲少';
teacher.age= 25;
teacher.subject = 'JavaScript';
//为教师对象添加行为方法
teacher.teach = function(){
    console.log('正在进行教学...');
};
```

上面的代码为教师对象添加了姓名、年龄和所教科目的属性，并且为其添加了一个教学行为teach的方法。

第二种，也可以直接通过字面值的方式来创建教师对象，示例如下：

```
//字面值直接创建对象
var teacher2 = {
    name:'Jaki',
    age:25,
    teach:function(){
        console.log('正在进行教学...');
    }
};
```

如果将teacher对象与teacher2对象的类型都进行打印，可以看到它们都是Object类型，例如：

```
console.log(teacher);          //{ name: '珲少', age: 25, subject: 'JavaScript', teach: [Function] }
console.log(teacher2);         //{ name: 'Jaki', age: 25, teach: [Function] }
console.log(typeof teacher);   //object
console.log(typeof teacher2);  //object
```

要使用对象的某个属性，有两种方式可以获取，比较方便且常用的方式是点语法，示例如下：

```
//取对象的属性
console.log(teacher.name);     //珲少
console.log(teacher.age);      //25
console.log(teacher.subject);  //JavaScript
```

也可以通过键名字符串的方式来获取对象的属性，示例如下：

```
//通过键名字符串取值
console.log(teacher['name']);        //珲少
```

需要注意，通过键名字符串的方式来取对象的属性时，所传入的键必须是字符串类型的。

对象中定义的函数用来描述对象的行为，同样可以使用点语法来使对象执行行为，示例如下：

```
//让对象执行行为
teacher.teach();               //将打印 正在进行教学...
```

同样，使用键名获取到的方法也可以执行：

```
teacher['teach']();            //将打印 正在进行教学...
```

至此，我们可以简单理解，万事万物都可以作为对象（但并不是全部）。基本的数据组合为简单的对象，简单的对象组合成复杂的对象，复杂的对象协作完成复杂的功能。这里，对于“对象”我们不做深入的研究，后面会详细介绍更加复杂的面向对象机制。

2-12　算术运算符

运算符是用来执行程序代码运算的。一个完整的表达式应该由两部分组成：操作数与运算符。例如，简单的加法表达式“1+2”中，数字“1”和数字“2”都是操作数，符号“+”是加法运算符，其作用是将前后两个操作数进行加法运算。

在ECMAScript中，运算符可以分为如下几类：

（1）算术运算符。

（2）赋值运算符。

（3）关系运算符。

（4）逻辑运算符。

（5）位运算符。

（6）自增、自减、条件、逗号等特殊运算符。

其中，算术运算符用来做常见的数学运算，例如加、减、乘、除等。符号“+”是ECMAScript中的加法运算符，数字或者字符串都可以使用“+”运算符进行相加操作。示例如下：

```
//加法运算中的几个特殊规则
console.log(1+NaN);                  //NaN
console.log(Infinity+Infinity);      //Infinity
```

```
console.log(-Infinity + -Infinity)          //-Infinity
console.log(1+'1');                         //11
```

在数学中，与加法互为逆运算的是减法，JavaScript中使用符号“-”来进行减法运算，示例如下：

```
//减法运算符
var sub = 10-5;
console.log(sub);       //5
```

减法运算符有一点非常特殊，如果进行减法运算的两个操作数中有字符串类型，且其中的字符串类型可以转换为数字，则JavaScript会自动将其转换为数字再进行减法运算。但如果其中有字符串不能转换为数字，则计算结果为NaN，示例如下：

```
console.log("10"-5);          //5
console.log("10"-"3");        //7
console.log("s"-3);           //NaN
console.log("10"-"a");        //NaN
```

针对一些特殊值的减法运算，JavaScript中也定义了一些规则，如下：

（1）如果某个操作数是NaN，则运算的结果为NaN。

（2）正无穷值减去正无穷值，结果为NaN。

（3）负无穷值减去负无穷值，结果为NaN。

（4）正无穷值减去负无穷值，结果为正无穷值。

（5）负无穷值减去正无穷值，结果为负无穷值。

示例如下：

```
//减法运算中的几个特殊规则
console.log(1-NaN);                         //NaN
console.log(Infinity-Infinity);             //NaN
console.log(-Infinity - -Infinity);         //NaN
console.log(Infinity - -Infinity);          //Infinity
console.log(-Infinity - Infinity);          //-Infinity
```

当符号“+”与符号“-”作为一元运算符时，它就成了正号运算符与负号运算符。对数字进行正号或负号运算时，正号运算会保持数字的正负性，负号运算会改变数字的正负性，示例如下：

```
console.log(+num1);         //不改变符号 10
console.log(+num2);         //不改变符号 -10
```

```
console.log(-num1);          //改变符号 -10
console.log(-num2);          //改变符号 10
```

正负运算符还有一个很实际的用途，即可以将字符串值强制转成数字值，这在开发中十分常用。示例如下：

```
console.log(typeof +"1")          //number
```

ECMAScript中使用乘法运算符“*”来进行乘法运算，示例如下：

```
//乘法运算符
var mul = 3*4;
console.log(mul);                 //12
```

对于乘法运算，也存在下面几条特殊的规则：

（1）如果某个操作数是NaN，则结果为NaN。
（2）无穷值乘以0，结果为NaN。
（3）无穷值乘以0以外的其他数字，结果为无穷值。
（4）无穷值乘以无穷值，结果为无穷值。

示例代码如下：

```
//乘法运算中的几个特殊规则
console.log(1*NaN);                      //NaN
console.log(Infinity*0);                 //NaN
console.log(Infinity * 1);               //Infinity
console.log(Infinity * -1);              //-Infinity
console.log(Infinity * Infinity);        //Infinity
console.log(-Infinity * -Infinity);      //Infinity
console.log(-Infinity * Infinity);       //-Infinity
```

运算符“/”在ECMAScript中用来进行除法运算，示例如下：

```
//除法运算符
var del = 88/10;
console.log(del);         //8.8
```

除法运算符对于特殊值运算的规则如下：

（1）如果某个操作数是NaN，则结果为NaN。
（2）无穷值除以无穷值，结果为NaN。
（3）无穷值除以非无穷值，结果为无穷值。
（4）非无穷值除以无穷值，结果为0。

（5）任何数除以0，结果为无穷值。

（6）0除以任何数，结果为0。

示例代码如下：

```
//除法运算中的几个特殊规则
console.log(10/NaN);              //NaN
console.log(Infinity/Infinity);   //NaN
console.log(Infinity/100);        //Infinity
console.log(10/Infinity);         //0
console.log(100/0);               //Infinity
console.log(0/100);               //0
```

ECMAScript中还支持求余运算，取余运算也叫取模运算。符号“%”为取模运算符。示例如下：

```
//取模运算符
var res = 17%8;
console.log(res);        //1
var res2 = 10.7%1.5
console.log(res2);       //约等 0.2
```

对于特殊值的取模运算，有如下规则：

（1）无穷值对任何值取模结果都是NaN。

（2）非无穷值对无穷值取模结果为非无穷值本身。

（3）0对任何数取模结果都是0。

（4）任何数对0取模结果都是NaN。

示例代码如下：

```
//取模运算中的几个特殊规则
console.log(Infinity%1);          //NaN
console.log(Infinity%Infinity);   //NaN
console.log(100%Infinity);        //100
console.log(0%100);               //0
console.log(100%0);               //NaN
```

需要注意，在很多编程语言中，取模运算都不可以以浮点数作为操作数。JavaScript是一种相对特殊的语言，它并没有对浮点数的取模运算做太严格的控制。

2-13　赋值运算符

赋值运算符的作用是将表达式的值赋给变量。从接触到ECMAScript语言开始，我们就一直在使用赋值运算符，最简单的赋值运算符使用示例如下：

```
//赋值运算符
var string = "Hello World";
```

ECMAScript中还提供了一些复合赋值运算符，示例如下：

```
//复合运算符
//复合加赋值运算符
var v1 = 0;
v1+=10;              //相当于 v1=v1+10;
console.log(v1);     //10
v1-=9;               //相当于 v1=v1-9;
console.log(v1);     //1
v1*=10;              //相当于 v1=v1*10;
console.log(v1);     //10
v1/=10;              //相当于 v1=v1/10;
console.log(v1);     //1
v1&=0;               //相当于 v1=v1&0;
console.log(v1);     //0
v1|=1;               //相当于 v1=v1|1;
console.log(v1);     //1
v1<<=1;              //相当于 v1=v1<<1;
console.log(v1);     //2
v1>>=1;              //相当于 v1=v1>>1;
console.log(v1);     //1
v1>>>=1;             //相当于 v1=v1>>>1;
console.log(v1);
```

其中，“&”“|”“<<”等符号可能看上去有些陌生，这些是ECMAScript中的位运算符，后面会专门介绍位运算的相关知识，这里你只需要记住，复合赋值运算符实际上是将一个变量作为操作数，经过计算后再赋值给它自身。

2-14　关系运算符

在代码的编写过程中，比较操作十分常用，例如比较两个字符串是否相同、比较两个数字的大小等。比较运算符的计算结果将会返回一个布尔值，通过布尔值的真或假可以实现不同的业务逻辑。

数字之间的比较是最常规的比较，示例如下：

```
//数字之间的比较
console.log(1<2);            //true
console.log(1>2);            //false
console.log(1==2);           //false
```

符号"<"为小于运算符，当第1个操作数小于第2个操作数时，结果为true，否则结果为false。">"为大于运算符，当第1个操作数大于第2个操作数时，结果为true，否则结果为false。需要额外注意，在ECMAScript中，等于运算符用符号"=="来表示，这和数学中的"="有些差异，初学者容易混淆，需特别注意。

比较运算符也可以用于字符串与字符串之间的比较操作，字符串的比较遵守这样的法则：逐字符进行字符码大小的比较，如果字符码相同，就比较下一个字符，直到比较出结果或者比较完所有字符。示例如下：

```
//字符串与字符串进行比较
console.log("a">"b");            //false
console.log("a"<"b");            //true
console.log("ss"=="ss");         //true
```

需要特别注意，如果是描述数字的字符串，比较时依然会遵守上面的法则，例如：

```
console.log("12">"3");           //false
```

"12">"3"的比较结果返回的是false，这是正确的。因为JavaScript解释程序会将字符"1"与字符"3"的字符码进行比较，将结果返回。

数字和字符串进行比较相对要棘手些。首先，如果是描述数字的字符串与数字进行比较，JavaScript解释程序会将字符串强制转换成数字类型后再进行比较，例如：

```
console.log("3">10);             //false
```

如果是非数字的字符串与数字进行比较，结果将永远是false。

```
console.log("a">0);              //false
```

ECMAScript中还可以使用“>=”与“<=”进行不小于和不大于的比较运算，其规则和“>”符号与“<”符号一致。示例如下：

```
console.log(12<=12);        //true
console.log(1>=2);          //false
```

关于等于与不等于的比较，在ECMAScript中有两类：一类是相等比较“==”与不相等比较“!=”；另一类是全等比较“===”与不全等比较“!==”。全等比较运算并非是ECMAScript语言所独有的，许多编程语言中都有类似的运算符，例如Swift。

在进行相等或不相等比较时，不同类型间数据的比较遵守如下几条原则：

（1）布尔值在比较运算前会被转换成数值，true转换成1，false转换成0。
（2）描述数字的字符串与数字进行比较前会被转换成数字。
（3）对象和字符串进行比较前，会将对象转换成字符串"[object Object]"。
（4）null值和undefined值进行相等比较，结果为true。

示例代码如下：

```
console.log(true==1);                    //true
console.log(2==true);                    //false
console.log(false==0);                   //true
console.log("11"==11);                   //true
var obj = {name:'jaki'};
console.log(obj=="[object Object]");     //true
console.log(1!=2);                       //true
console.log(null==undefined);            //true
```

需要注意，如果进行比较操作的是引用值而非原始值，则比较的实际是所引用对象的地址。

再次提醒，NaN与NaN进行相等比较，结果是false。

“==”与“!=”运算在进行比较前，会根据上面的规则对操作数进行类型的转换，全等运算符“===”与不全等运算符“!==”在比较前不会做任何类型转换，换句话说，全等和不全等进行比较时，既会比较类型，又会比较值，只有类型和值完全相等的两个操作数才被认为是全等。示例如下：

```
//全等比较
console.log(11==="11");     //false
console.log(true!==1);      //true
```

有一个很有趣的小例子，在JavaScript中，如果下面的代码输出false：

```
console.log(ex>1);
```

那么如下代码将一定输出true么?

```
console.log(ex<=1);
```

答案是否定的，如果ex变量在进行比较转换时被转换成了NaN，那么上面两句输出的都将是false：

```
var ex = "ss";
console.log(ex>1);            //false
console.log(ex<=1);           //false
```

2-15　逻辑运算符

逻辑运算对于一门编程语言至关重要，它是分支和循环结构的基础，ECMAScript中支持的逻辑运算有3种：逻辑与运算、逻辑或运算和逻辑非运算。

ECMAScript中使用符号“&&”进行逻辑与运算。逻辑运算通常在两个布尔类型的操作数之间进行，“与”运算需要遵守表2-3所示的运算规则。

表2-3　“与”运算符的运算规则

操作数 1	操作数 2	结　果
true	true	true
true	false	false
false	true	false
false	false	false

上面的运算规则可以简要概述为：进行逻辑与运算的两个操作数都为true，结果才为true，只要有一个操作数为false，结果就为false。

在有些强类型的编程语言中，逻辑运算符只能在布尔值之间进行运算，在ECMAScript中，逻辑运算的操作数可以是任意类型的，并且其运算结果也不一定是布尔类型的值，规定如下：

（1）在两个对象间进行逻辑与运算，结果将返回第二个对象。

（2）在进行逻辑与运算的两个操作数中，如果有一个操作数为null，则结果为null。

（3）在进行逻辑与运算的两个操作数中，如果有一个操作数为NaN，则结果为NaN。

（4）在进行逻辑与运算的两个操作数中，如果有一个操作数为undefined，则结果为undefined。

示例代码如下：

```
//与运算的相关规则
var obj = {name:'jaki'};
//两个对象进行逻辑与运算，结果为第二个对象
console.log({}&&obj);
console.log(null&&true);          //null
console.log(true&&null);          //null
console.log(NaN&&true);           //NaN
console.log(true&&NaN);           //NaN
console.log(undefined&&true);          //undefined
console.log(true&&undefined);          //undefined
```

下面来看一个十分有趣的小例子：

```
var v1 = 10;
var v2 = true;
console.log(v2&&(v1++));
console.log(v1);
var v3 = 10;
var v4 = false;
console.log(v4&&(v3++));
console.log(v3);
```

你能猜出上面代码中的console.log(v1)与console.log(v3)分别会打印出什么样的结果吗？

结果是console.log(v1)将打印出11，而console.log(v3)将打印出10。

对上面的结果是不是有些意外？其实很多编程语言在处理逻辑运算时都有这样一种法则：如果第一个操作数已经可以确定此表达式的结果，则不会再执行第二个操作数。从上面的代码来看，v4为false时，已经可以确定此与运算结果为false，因此v3++将不会被执行到。

ECMAScript中使用符号“||”进行逻辑或运算。逻辑或运算遵守表2-4所示的运算规则。

表2-4　“||”运算符的运算规则

操作数 1	操作数 2	结　果
true	true	true
true	false	true
false	true	true
false	false	false

和逻辑与运算一样，ECMAScript中的逻辑或运算也不一定会返回逻辑值。规定如下：

（1）如果有一个操作数为对象，当对象为第1个操作数时，结果为对象本身；当对象为第2个操作数时，如果第1个操作数为false，则结果为对象本身，如果第1个操作数为true，则结果为true。

（2）如果两个操作数都为对象，则返回第一个操作数。

示例代码如下：

```
//或运算规则
console.log(obj||false);        //obj
console.log(true||obj);         //true
console.log({}||obj);           /{}
```

因此，在进行逻辑或运算时，如果第一个操作数已经可以决定表达式的值，则不会再执行到第二个操作数处。

ECMAScript中的逻辑非运算使用“！”符号定义，需要注意，逻辑非运算一定会返回布尔值。逻辑非运算也被称为逻辑取反运算，其遵守表2-5所示的运算规则。

表2-5 “！”运算符的运算规则

操作数	结　果
True	false
False	true

当操作数不全是逻辑值时，有如下规则：

（1）如果操作数是对象，则返回false。

（2）如果操作数是数字0，则返回true。

（3）如果操作数是非0数字，则返回false。

（4）如果操作数是null，则返回true。

（5）如果操作数是NaN，则返回true。

（6）如果操作数是undefined，则返回true。

2-16　位运算符

我们知道，程序中的所有数在计算机内存中都是以二进制的形式存储的。这很好理解，进制的实质是确定计数时逢几进一。人类有10只手指，因此很久以前，祖先就习惯了使用十进制来计数。计算机的核心是由电子元件组成的，而电子元件最容易描述的两种状态是高电平与低电平，因此使用二进制计数是最安全、最便捷的方式。

在介绍ECMAScript中的位运算前，先来简单地了解一下JavaScript中二进制计数。JavaScript中只有一种数值类型：原始类型Number。但是实际上，JavaScript中存储的数值有两种，分别为有符号数和无符号数。其实这和大多数编程语言类似，只是有些强类型的语言会将数值类型再进行细化，比如8位整型、32位整型、64位整型、32位浮点型或64位浮点型。JavaScript中所有的数值默认都是32位的（当然，这样说并不准确，具体的位数和计算机环境有关，目前大多都是32位的）。位数即表示一个数字需要多少个二进制位，我们暂定JavaScript中所有的数值都是32位的，那么8这个十进制数在内存中存储的数据如下：

0	0	……	…..	0	0	1	0	0	0

上面每一个方格表示一个二进制位，中间的省略号代表省略中间的0，一共32个小方格，代表32个数位。

JavaScript中所有的数值创建时默认都是有符号的，虽然存储一个数值需要32位，我们能够操作的实际上只有31位，最后一位作为符号位，符号位为0表示这个数值是正数，符号位为1表示这个数值是负数。对于正数，存储在内存中的二进制数据很好理解，将此正数的二进制形式放入内存，其余位补零即可。但是对于负数，其在内存中是以二进制补码方式存储的，计算补码的步骤如下：

（1）确定该数的绝对值的二进制形式。

（2）对此二进制码求反码（0和1互相交替）。

（3）在反码的基础上加1。

根据上面的规则，以十进制数-8为例，其绝对值的二进制形式为0…01000，对其求反码为1…10111，在其基础上再加1得到1…11000，即十进制数-8实际存在内存中的数据如下：

1	1	……	……	1	1	1	0	0	0

相对于有符号数而言，无符号数中并没有负数，所有的数值都是正数，在这种情况下，正负位就失去了作用，因此对于无符号数来说，其32个二进制位都用来表示数字。

额外说一点，计算机中为什么要采用补码的方式来存储数据呢？对于有符号数，最高位表示的是符号，如果直接进行二进制形式的存储，难免会出现这样一种情况：0可以表示为正数0和负数0，这有悖现实规律。因此，人们采用补码的方式使现实的数值与计算机内存中存储的二进制数据一一对应，正数的补码是其本身，负数的补码是其反码加1。经过这样的计算后，无论是正数0还是负数0，在计算机内存中存储的都是全0码，做到了统一。

理解了计算机中的二进制计算原理，我们再来看位运算符。顾名思义，位运算就是在二进制位的基础上进行运算，其直接对二进制位进行操作。ECMAScript中支持的位运算有7种，分别为按位非运算、按位与运算、按位或运算、按位异或运算、按位左移运算、按位有符号右移运算和按位无符号右移运算。

按位非运算使用符号“~”来定义，它也被称为按位取反运算，即原二进制位是1的变为0，原二进制位是0的变为1。示例代码如下：

```
var v3 = ~8;
console.log(v3);          //-9
```

对8进行按位取反运算后，结果将为-9，如果将十进制换成二进制表示，这个过程就很好理解，首先8的二进制形式如下：

0	0	……	……	0	0	1	0	0	0

按位取反后如下：

1	1	……	……	1	1	0	1	1	1

前面说过，负数存储的实际上是补码，那么通过逆运算，先对补码减1，如下：

1	1	……	……	1	1	0	1	1	0

再对补码减1后得到的反码取反，如下：

0	0	……	……	0	0	1	0	0	1

上面的原码就是我们最终结果的绝对值形式，将其转换成十进制并且加上负号，就得到了-9。

在编程中，你并不需要对每一次按位取反操作都进行如上推演，上面介绍的过程只是原理，理解了原理后，我们可以通过技巧记忆的方式来快速得到想要的答案，即对数值的按位取反操作实际上就是将此数值求负再减1。

按位与运算使用“&”符号定义，是一个二元运算符，其进行运算的两个操作数的对应二进制位分别进行与运算后将结果返回，即如果进行运算的相应位都为1，最终结果数值的此二进制位为1，否则为0。示例代码如下：

```
var v4 = 1&9;
console.log(v4);          //1
```

分解上述代码的计算过程如下。

1的二进制码：

0	0	……	……	0	0	0	0	0	1

9的二进制码：

0	0	……	……	0	0	1	0	0	1

进行按位与运算后：

0	0	……	……	0	0	0	0	0	1

最终结果为1。

按位或运算使用“|”符号定义，是一个二元运算符，其进行运算的两个操作数的对应二进制位分别进行或运算后将结果返回，即如果进行运算的相应位都为0，最终结果的此二进制位为0，否则为1。示例代码如下：

```
var v5 = 8|3;
console.log(v5);        //11
```

分解上述代码的计算过程如下。

8的二进制码：

0	0	……	……	0	0	1	0	0	0

3的二进制码：

0	0	……	……	0	0	0	0	1	1

进行按位或运算后：

0	0	……	……	0	0	1	0	1	1

最终结果为11。

按位异或运算使用符号“^”定义，是一个二元运算符，其进行运算的两个操作数对应二进制位分别进行异或运算后将结果返回，即如果进行运算的相应位不同，最终结果数值的此二进制位为1，否则为0。示例代码如下：

```
var v6 = 8^11;
console.log(v6);        //3
```

分解上述代码的计算过程如下。

8的二进制码：

0	0	……	……	0	0	1	0	0	0

11的二进制码：

0	0	……	……	0	0	1	0	1	1

进行按位异或运算后：

0	0	……	……	0	0	0	0	1	1

最终结果为3。

按位左移运算使用符号“<<”定义，其作用是将二进制数据向左移动指定的位数，右侧空出来的位将进行补零操作。需要注意，按位左移操作并不会影响符号位，移动过程并不包括符号位，示例代码如下：

```
var v7 = -2<<2;
console.log(v7);        //-8
```

分解上述代码的计算过程如下。

-2的二进制码（补码）：

1	1	……	……	1	1	1	1	1	0

进行左移两位的运算后：

1	1	……	……	1	1	1	0	0	0

对补码求原码：

0	0	……	……	0	0	1	0	0	0

最终结果为-8。

与按位左移运算相对应的还有按位右移运算。需要注意，按位右移运算有两种：有符号按位右移运算与无符号按位右移运算。其中，有符号按位右移运算与按位左移运算互为逆运算，使用“>>”符号定义，示例如下：

```
var v8 = -8>>2;
console.log(v8);        //-2
```

无符号按位右移运算和有符号按位右移运算最大的不同在于：无符号按位右移运算时，并不保留符号位，会将符号位一起进行移动，其使用“>>>”符号定义。正数的符号位为0，因此对正数并没有影响，负数就不同了，示例如下：

```
var v9 = 8>>>2;
console.log(v9);            //2
var v10 = -8>>>2;
console.log(v10);           //1073741822
```

具体过程这里不再重复，你可以根据前面的示例自行推导一下。

因此，在使用无符号右移运算时要极其小心。

2-17　自增与自减运算符

C语言中定义了自增与自减两种运算符，它们是很多初学者的噩梦。你或许猜到了，ECMAScript中也定义了这两种运算符，并且和C语言中定义的用法基本一致。

自增运算符使用符号“++”定义，自减运算符使用符号“--”定义。简单理解，自增运算符是在操作数本身的基础上进行加1运算，自减运算符是在操作数本身的基础上进行减1运算，示例代码如下：

```
//自增与自减运算符
var a = 10;
var b = 10;
//进行自增与自减运算
a++;
b--;
console.log(a);          //将打印 11
console.log(b);          //将打印 9
```

需要注意，自增和自减运算符可以放在操作数后面，也可以放在操作数前面。如果将运算符放在操作数后面，通常称其为“后置自增/减运算符”；如果将运算符放在操作数前面，通常称其为“前置自增/减运算符”。“前置”与“后置”虽然只是一字之差，其运算过程与结果却差别很大。

先来看下面这个例子：

```
//自增/减运算符的前置与后置
var c = 10;
var d = 10;
console.log(c++);          //将打印 10
console.log(++d);          //将打印 11
console.log(c);            //将打印 11
console.log(d);            //将打印 11
```

单独打印变量c和变量d的结果都将是11，说明无论是前置自增运算还是后置自增运算，都是在原操作数的基础上进行加1运算。然而如果对“c++”和“++d”这两个表达式的返回值进行打印，可以发现前置自增运算返回的是运算完成后的值，而后置自增运算返回的是运算前的值。同样的规则也适用于自减运算符，示例代码如下：

```
var e = 10;
var f = 10;
console.log(e--);        //将打印 10
console.log(--f);        //将打印 9
console.log(e);          //将打印 9
console.log(f);          //将打印 9
```

2-18　条件运算符

在开发中，条件语句的编写必不可少，然而最简单的条件结构也需要至少3行功能代码，示例如下：

```
//条件结构
var res;
if(true){
    res = 10;
}else{
    res = 0;
}
console.log(res);        //将打印 10
```

JavaScript中提供了条件运算符“?:”来简化表达的条件结构。上面的示例可以简化成如下代码：

```
//条件表达式
var res = true?10:0;
console.log(res);        //将打印 10
```

条件运算符组成的表达式结构为“逻辑值?表达式1:表达式2”，当问号前面的逻辑值为true时，运算结果为表达式1的值，当问号前面的逻辑值为false时，运算结果为表达式2的值。

2-19　逗号运算符与 delete 运算符

ECMAScript中还定义了一种逗号运算符，其作用是将多个表达式放入一行语句中执行，示例如下：

```
//逗号表达式
var r1 = 1+3,r2=1*3;
console.log(r1),console.log(r2);          //将打印 4 3
```

逗号运算符在进行多个变量的声明时，十分方便。

JavaScript中还提供了一个十分特殊的运算符“delete”，“delete”运算符用于将对象中的某个属性删除，示例如下：

```
var obj = {
    name:"珲少",
    age:25
};
console.log(obj.name);       //将打印珲少
delete obj.name;
console.log(obj.name);       //将打印 undefined
```

关于对象的更多内容，后面章节会详细介绍，这里你只需要了解“delete”运算符的作用即可。

2-20　关于运算符的优先级与结合性

在任何编程语言中，运算符的优先级与结合性都是一个老生常谈的话题。小学数学老师都一遍遍地告诉过我们“先乘除，后加减”的法则。在ECMAScript语法中，也遵守类似的法则。例如如下表达式计算的值是22而不是28：

```
var res =   2+5*4;
console.log(res);            //结果为 22
```

所谓运算符的优先级，是指不同运算符在同一个表达式中执行运算的先后顺序。优先级高的运算符将优先被执行运算，例如上面示例代码中的“*”运算符的优先级要高于“+”运算符，因此先进行乘法运算，再进行加法运算。

除了“优先级”的概念外，运算符还有“结合性”概念。对于优先级相同的运算符，“结合性”决定了其表达式中运算的执行顺序，结合性分为左结合性和右结合性，左结合性的运算符将从左向右依次执行，右结合性的运算符将从右向左依次执行，示例如下：

```
//结合性
//左结合性
var a = 1+2+3;          //结果为 6，相当于(1+2)+3
```

```
//右结合性
var b = c = 5;            //相当于 c=5; b=c;
```

常用运算符的优先级与结合性如表2-6所示。

表2-6　运算符的优先级与结合性

运算符	优先级	结合性
小括号：()	19	-
后置递增：++	16	-
后置递减：--	16	-
逻辑非：!	15	右
按位非：~	15	右
正号运算符：+	15	右
负号运算符：-	15	右
前置递增：++	15	右
前置递减：--	15	右
Delete	15	右
乘法：*	14	左
除法：/	14	左
取模：%	14	左
加法：+	13	左
减法：-	13	左
按位左移：<<	12	左
按位右移：>>	12	左
按位无符号右移：>>>	12	左
小于：<	11	左
小于等于：<=	11	左
大于：>	11	左
大于等于：>=	11	左
等于：==	10	左
非等：!=	10	左
全等：===	10	左
非全等：!==	10	左
按位与：&	9	左
按位异或：^	8	左
按位或：|	7	左
逻辑与：&&	6	左

（续表）

运算符	优先级	结合性
逻辑或：‖	5	左
条件：?:	4	右
赋值：=	3	右
逗号：,	0	左

来看一个小例子，你能猜出下面代码的计算结果吗？

```
//例子
var i=3;
var j=3;
var n=3;
var a = i++ + i++ + i++;            //3+4+5
var b = ++j + ++j + ++j;            //4+5+6
var c = n++ + ++n + n++;            //3+5+5
console.log(""+i+" "+j+" "+n+" "+a+" "+b+" "+c);        //6,6,6,12,15,13
```

无论你对运算符的优先级与结合性记忆如何，给你一个建议：如果有控制运算顺序的必要，请强制使用小括号，一目了然，省时省心。

2-21 隐式类型转换

不夸张地说，类型转换在ECMAScript中无时无刻不在进行。有些类型转换浅显易见，例如使用内置函数来进行类型转换，代码如下：

```
var n1 = 5;
var s1 = String(n1);
//数值转换成字符串
console.log(s1,typeof s1);            //5 string
```

这种使用函数手动进行类型转换的方式常常被称为显式类型转换，显式转换一般不会为代码带来风险。作为开发者，我们可以一眼看出转换前后的类型。在ECMAScript中，最令开发者提心吊胆的是隐式转换，稍不留神就会掉入其中的陷阱。

当你将数字与字符串进行相加操作时，数字会被隐式转换成字符串再进行加操作，这在前边的案例中也有介绍。其实在ECMAScript中，对象之间的加操作也会被隐式转换成字符串，例如：

```
var obj1 = {name:"Jaki"};
var obj2 = {age:25};
var v1 = obj1+obj2;
console.log(v1);                //[object Object][object Object]
```

加法运算符虽然会经常产生隐式转换，但是其并不是真正的雷区。真正的雷区是关系运算符。前面介绍过，在进行相等比较时，ECMAScript中提供了两种运算符，一种是相等运算符“==”，另一种是全等运算符“===”。前面我们说全等运算符要求除了值相等外，类型也要相等。其实这是一种不正确的说法（只是因为便于理解记忆并且比较流行，所以我们姑且这么说），相等与全等运算符的真正区别在于相等运算符会进行隐式类型转换，而全等运算符不会。第一种说法会让你误认为全等运算符做了更多工作：既比较值又比较类型。实际上恰好相反，相等运算符做的工作更多：比较前先进行隐式类型转换。我们来看一个简单的命题：

有变量a，a == !a一定不成立。

这个命题乍看上去必然为真，那么我们通过代码来测试一下：

```
var a="0";
var res = (a == !a);
console.log(res);       //true
```

上面的打印结果为true，你一定目瞪口呆，一个值的取反竟然和它本身是相等的。如果你了解隐式类型转换的过程，这个神奇的结果其实一点也不神奇。首先，对字符串"0"进行逻辑非运算时，会被转换成逻辑值false，进行等于比较运算时，逻辑值false会被隐式类型转换为数值0，而字符串"0"也会被隐式转换成数值0，因此比较的结果为true。相等运算符在进行不同类型间的比较时，大部分情况都会朝数值的方向进行隐式类型转换，在使用时一定要格外注意。

还有一些经常会令开发者疑惑的情况，请看下面的命题：

（1）有变量a和b，且都不等于NaN，则a>b、a<b和a==b中一定有一个是成立的。
（2）有变量a和b，且都不等于NaN，则a>=b和a<=b中一定有一个是不成立的。

这两个命题看上去都是无懈可击的，看过如下代码，你就会改变想法了：

```
var a = {};
var b = {};
console.log(a>b);           //false
console.log(a<b);           //false
console.log(a==b);          //false
```

```
console.log(a>=b);          //true
console.log(a<=b);          //true
```

奇怪吧，上面的代码前3个全部打印了false，当对“对象”进行大于或小于比较时，会将其隐式转换为字符串，对象转换成字符串后都为[object Object]，因此前两个log语句都打印了false。而针对a==b的比较，这里并没有进行隐式类型转换，回忆一下前面讲解过的值类型与引用类型就能豁然开朗，对于对象只有其引用完全相同时，才算相等。

第2个命题也是假的，后两个打印了true，击破这个命题的原因在于ECMAScript中对“>=”和“<=”两种运算符的运算会自动被转换成“<”和“>”。

隐式类型转换是ECMAScript中的一把双刃剑，其给开发者编写代码带来便利的同时，也存在很多隐患，在编写代码时一定要多多注意。

2-22　编程练习

练习1：预测下面log语句输出的值。

```
var a = 3;
var b = {age:26};
var c = a;
c = 4;
var d = b;
d.age = 25;
console.log(a,c,b.age,d.age);
```

解析：将会输出：3 4 25 25。本练习主要考察对JavaScript中值类型和引用类型的理解。值类型数据直接存在变量所在的内存中，在赋值时会直接复制原始值，引用类型变量中存放的是数据所在的地址，赋值时赋的是地址，因此在修改时会影响所有的变量。

练习2：编写一个函数，对传入的参数进行类型检查，如果为undefined或者null，返回布尔值假，否则返回真。

解析：

```
function check(param){
    if (param == undefined || param == null) {
        return false;
    }
    return true;
}
```

练习3：分别用八进制和十六进制和科学计数法来表示十进制数99。

解析：

科学计数法：9.9e1。

八进制：0143或0o143。

十六进制：0x63。

练习4：编写函数，实现如下功能。传入两个字符串参数，以换行符将两个字符串进行拼接，之后返回。

解析：

```
function func(s1,s2){
    return s1+'\n'+s2;
}
console.log(func("你好","JavaScript"));
```

练习5：用两种方式创建教师对象，为其添加一个name属性，并用两种方式来进行属性的访问。

解析：

```
var teacher1 = {
    name:'Jaki'
}
var teacher2 = new Object();
teacher2.name = 'Lucy';
console.log(teacher1.name,teacher2['name']);        //Jaki Lucy
```

练习6：创建一个函数，其功能是生成1～100间的一个随机数。

解析：

```
function rand(){
    return Math.floor(Math.random()*100);
}
console.log(rand());
```

floor是JavaScript中的一个数学函数，用来进行浮点数的向下取整，random函数用来生成一个0到1之间的随机浮点数。

练习7：思考一下，如何编写一个函数，不使用乘法运算符来实现乘以2的n次方运算。

解析：

```
function func(param,c){
```

```
        return param<<(c);
}
console.log(func(3,2));       //3*2^2 = 12
```

由于二进制数的运算特点，因此使用左移位运算可以快速实现乘以2的n次方运算。

练习8：编写一个函数，实现如下逻辑。若传入的参数为布尔值，则进行取反后返回；若传入的是字符串值，则在前面拼接“hello，string：”后返回；如果是大于100的数值，就返回100，不大于100的数值则返回1；如果是对象，就返回字符串“Object”；其他情况均返回数值0。

解析：

```
function func(param){
        if (typeof param === 'boolean') {
                return !param;
        }
        if (typeof param === 'string') {
                return "hello,string:"+param;
        }
        if (typeof param === 'number') {
                if (param>100) {
                        return 100;
                }else{
                        return 1;
                }
        }
        if (typeof param === 'object') {
                return "Object";
        }
        return 0;
}
```

练习9：编写函数，使用条件运算符实现如下逻辑。输入的参数能整除3，就返回true，否则返回false。

解析：

```
function func(param){
        return param%3==0?true:false;
}
console.log(func(1));         //false
```

练习10：你能看出下面log语句的输出吗？

```
console.log(1=='1');
console.log(1==='1');
console.log(NaN===Infinity);
console.log(Infinity == Infinity+1);
console.log(NaN == NaN);
console.log({}=={});
```

解析：

```
console.log(1=='1');          //true
console.log(1==='1');         //false
console.log(NaN===Infinity);          //false
console.log(Infinity == Infinity+1);          //true
console.log(NaN == NaN);          //false
console.log({}=={});          //false
```

“==”运算符和“===”运算符的最大区别是“==”会进行隐式类型转换。

第 3 章

ECMAScript 流程控制与函数

程序的逻辑都是由流程来实现的，任何一种编程语言都要有控制程序流程的相关功能。说到程序流程，最重要的两种结构就是分支结构与循环结构。分支结构使程序有了灵活的选择能力，循环结构则使程序有了重复大量工作的能力。当然，分支、异常、异步等结构也是ECMAScript中重要的流程控制方式。

3-1　if-else 分支结构

if-else语句是JavaScript中最常用的条件语句。使用if语句，开发者可以根据不同的条件做不同的逻辑处理。最简单的if语句示例如下：

```
//if 结构 1
var condition = 10>5;
if (condition) {
	console.log("分支一");		//将执行
}
console.log("结束");
```

if关键字后面的小括号中需要写入一个要进行判断的条件，此表达式不一定是严格的Boolean类型，ECMAScript会自动对其进行Boolean类型转换。需要注意，为了减少歧义与出错率，建议在if条件中编写严格返回Boolean值的表达式。如果if条件最终为true，程序就会执行大括号中的代码块，如果if条件最终为false，程序就会跳过大括号中的代码块向后执行。

if-else语句还有额外两种结构，可以方便地进行多分支结构的处理，示例如下：

```
//if 结构 2
if (condition) {
        console.log("分支一");
}else{
        console.log("分支二");
}
console.log("结束");

//if 结构 3
if (condition) {
        console.log("分支一");
}else if(condition2){
        console.log("分支二");
}else if(condition3){
        console.log("分支三");
}else{
        console.log("分支四");
}
console.log("结束");
```

结构2与结构1的区别在于：如果if条件为false，结构1就会跳过if结构，而结构2则会执行else对应的大括号中的代码块。结构3是一种多条件分支结构，会依次判断if条件是否为true，遇到一个if条件为true后，则将执行其对应的代码块，并且其后的if结构都会被跳过。

3-2　switch-case 分支结构

在学习switch-case结构之前，你可以先思考一个简单的场景：学生综合成绩满分5分，最低1分。分数从高到低依次代表卓越、优秀、良好、及格和不及格。试编写ECMAScript程序来自动输出分数对应的档次。

使用if-else结构可以编写出如下代码：

```
var score = 4;
if (score==1) {
    console.log("不及格");
}else if(score==2){
    console.log("及格");
}else if(score==3){
    console.log("良好");
}else if(score==4){
    console.log("优秀");
}else if(score==5){
    console.log("卓越");
}else{
    console.log("无效的分数");
}
```

上面的示例代码可以完成题目的要求，但是大量if-else判断使得代码变得十分冗余，看上去并不简洁，使用switch-case结构可以很好地解决这一问题。

switch-case也是ECMAScript中的一种多分支结构，使用switch-case结构重新改写上面的代码如下：

```
switch (score) {
    case 1:{
            console.log("不及格");
        }
        break;
    case 2:{
            console.log("及格");
        }
        break;
    case 3:{
            console.log("良好");
        }
        break;
    case 4:{
            console.log("优秀");
        }
        break;
    case 5:{
            console.log("卓越");
        }
```

```
                break;
            default:{
                    console.log("无效的分数");
                }
        }
```

switch关键字后面需要指定一个表达式，case子句对应的值如果和此表达式的值相等，就会执行此case对应的代码块。需要注意，break语句用于跳出switch-case结构，即一旦某个case匹配成功，则不再运行后续逻辑代码。这点在开发中要根据实际情况选择是否使用break跳出，default块则是当所有case都匹配失败后执行的代码。

在许多编程语言中，switch-case结构都有一个特殊的要求，进行匹配的值必须为整型数据。JavaScript则不同，switch-case结构进行匹配的值可以是字符串，甚至是其他变量。

3-3　while 循环结构

循环结构又称迭代结构，用来多次重复地执行某一块逻辑代码。我们在计算数学题时，思路是向着简单化与公式化的，因为人的大脑有极限，无法也不可能进行超量的计算工作。但是计算机解决问题的思路与之刚好相反，计算机的运算速度非常快，一般情况下根本无须考虑运算性能的问题。因此，作为开发者，在编写代码时更应该考虑代码的简洁性与易读性。

如果有人问你从1依次递增100的连加结果是多少，我想你可能首先会想到等差数列的求和公式：首项加末项的和乘以项数除以二。在编程中要解决这个问题，使用循环结构更加简单。

下面的示例代码分别演示使用数学公式的方式和使用循环结构的方式对等差数列进行求和运算：

```
//数学公式进行等差数列的计算 1..100
var res = (1+100)*100/2;
console.log(res);        //5050
//使用循环结构来进行计算
var i=1;
var res2 = 0;
while(i<=100){
    res2+=i;
    i++;
}
console.log(res2);            //5050
```

上面的示例代码中使用到的while结构是ECMAScript中十分常用的一种循环结构，while关键字后面需要指定一个循环条件，当此条件成立时，会执行while循环体中的代码，执行完成后，会继续判断循环条件是否成立，如果条件成立，就会再次执行循环体，直到条件不成立为止。

while循环还有一种变体，其被称为do-while循环结构，与while循环的区别在于，do-while结构需要先执行一次循环体，之后进行循环条件的判断，如果条件成立，就会再次执行循环体，直到条件不成立为止。上面的求和运算使用do-while结构的方式改写如下：

```
//do-while 循环
i=1;
res3 = 0;
do{
	res3+=i;
	i++;
}while(i<=100);
console.log(res3);		//5050
```

需要注意，在编程中，无限循环也叫作死循环，一般情况下，我们要避免编写出死循环的代码。使用while结构编写最简单的无限循环如下：

```
while(true){
	console.log("...");
 }
```

3-4　for 循环结构

for循环结构是比while循环结构更加常用的一种循环结构，在许多流行编程语言中也是如此。for循环的基本编写格式如下：

```
for(init;exp;exc){}
```

其中，init语句对循环变量进行初始化，exp语句是循环的条件，exc语句用于修改循环变量，示例如下：

```
//for 循环
var res4 = 0;
for (var i = 1; i <= 100; i++) {
	res4+=i;
}
```

```
console.log(res4);            //5050
```

for循环语句的格式十分自由，init语句、exp语句甚至exc语句都是可以省略的。若已经存在循环变量，则init语句可以省略，示例如下：

```
var res4 = 0;
var i = 1;
for (; i <= 100; i++) {
        res4+=i;
}
console.log(res4);            //5050
```

同样，如果循环变量不需要修改，exc语句也可以省略，示例如下：

```
//for 循环
var res4 = 0;
var i = 1;
for (; i <= 100;) {
        res4+=i++;
}
console.log(res4);            //5050
```

需要注意，如果省略循环条件，for循环就会无限循环下去，除非遇到中断语句。用for循环结构编写最简单的无限循环如下：

```
for(;;)
```

3-5　关于 for-in 与 for-of 结构

ECMAScript中还提供了两种十分特殊的迭代结构：for-in与for-of。这两种结构都用于对象属性的枚举。到目前为止，你可能对对象的理解还不够深入，我们暂且不谈对象的更多内容，你只需要知道对象中可以封装属性，而for-in和for-of结构可以用来遍历对象的属性信息。

for-in用来遍历对象中属性的键，例如：

```
var teacher = {
        name:'jaki',
        age:24,
        subject:'JavaScript'
}
```

```
//将打印 name age subject
for (let key in teacher) {
    console.log(key);
}
```

for-of结构则更加特殊，其是ECMAScript 6引入的新特性。需要注意，for-of并不能作用于对象类型，是对集合的一种遍历，例如数组、集合、字符串等。示例代码如下：

```
var array = ["jaki","lucy","mery","jie"]
//将打印 jaki lucy mery jie
for (let value of array){
    console.log(value);
}
//将打印 h e l l o
for (let value of "hello"){
    console.log(value);
}
```

其实，for-of结构是配合ES6中的迭代器进行使用的，你也可以让teacher对象实现迭代器方法，示例如下：

```
var teacher = {
    _innerKeys:["name","age","subject"],
    _index:0,
    name:'jaki',
    age:24,
    subject:'JavaScript',
    [Symbol.iterator]: function () {
        return this;
    },
    next:function(){
        let obj = {value:this[this._innerKeys[this._index++]],done:this._index>
this._innerKeys.length};
        if (this._index>this._innerKeys.length) {
            this._index=0;
        }
        return obj;
    }
}
```

```
//将打印 jaki 24 javascript
for(let value of teacher){
        console.log(value);
}
```

关于迭代器的更多内容，后面会专门进行讲解。

3-6 break 中断语句

在ECMAScript中，中断语句有两种，分别为break语句与continue语句。中断语句与标签语句结合使用可以更加灵活地控制程序的执行流程。

在讲解switch-case结构时，我们已经使用过break语句，在switch-case结构中，break语句可以直接跳出当前的switch-case模块。同样，在循环结构中，也可以使用break语句来提前终止循环。示例代码如下：

```
//break 语句
for(var i=0;i<5;i++){
        console.log(i);          //依次输出 0、1、2、3
        if (i==3) {
                break;
        }
}
```

上面的代码中，如果不添加break语句，程序将依次输出0、1、2、3、4。添加了break语句后，当循环变量i自增到3时，程序将跳出循环结构，控制台只会输出0、1、2、3。需要注意，break语句默认将跳出当前所在的最内层循环。例如，下面的代码演示在多层循环中使用break语句：

```
for(var i=0;i<3;i++){
        console.log("i="+i);
        for(var j=0;j<3;j++){
                if (i==0) {
                        break;
                }
                console.log("j="+j);
        }
        console.log("=====");
}
```

```
/*
打印结果
i=0
=====
i=1
j=0
j=1
j=2
=====
i=2
j=0
j=1
j=2
=====
*/
```

从程序的打印结果可以看出，break语句跳出了内层循环，外层循环依然继续执行。

对于多层嵌套的循环，如果需要灵活控制跳出动作，可以将break语句与标签语句结合使用，示例如下：

```
label:for(var i=0;i<3;i++){
        console.log("i="+i);
        for(var j=0;j<3;j++){
                if (i==0) {
                        break label;
                }
                console.log("j="+j);
        }
        console.log("=====");
}
/*
打印结果
i=0
*/
```

从打印结果可以看出，上面代码中的break语句直接跳出了外层循环。标签语句的格式十分简单，在要加标签的语句前使用标签名加冒号的形式标注即可，break关键字后面可以指定一个已存在的标签名来跳出特定的循环。

需要注意，从原理上讲，循环结构是可以无限嵌套的，但是在实际开发中，代码的整洁与易读性也十分重要，编写循环结构时，应尽量保持不超过3层嵌套。中断语句与标签语句

结合使用时，可以极大地提高程序的灵活性。但同时也增加了代码的调试难度，建议如果不是必需的，尽量少用标签语句，如果一定要用，也应该使标签的命名易于理解。

3-7　continue 中断语句

continue语句也是CMAScript中常用的中断语句，其和break语句有着很大区别。break语句是跳出当前的循环结构，而continue语句则是跳出本次循环。示例如下：

```
//continue 语句
for(var i=0;i<5;i++){
    if (i==2) {
        continue;
    }
    console.log(i);          //依次输出 0，1，3,4
}
```

从结果可以看出，continue语句的作用是使程序仅跳过循环变量i等于2时的输出代码，并不会影响循环的继续执行。同样，对于多层循环结构，continue语句也默认只跳出内层循环的本次循环，示例如下：

```
for(var i=0;i<3;i++){
    console.log("i="+i);
    for(var j=0;j<3;j++){
        if (j==0) {
            continue;
        }
        console.log("j="+j);
    }
    console.log("=====");
}
/*
打印结果
i=0
j=1
j=2
=====
i=1
j=1
```

```
j=2
=====
i=2
j=1
j=2
=====
*/
```

continue语句也可以和标签语句结合使用来跳出指定循环结构的本次循环，示例如下：

```
label2:for(var i=0;i<3;i++){
    console.log("i="+i);
    for(var j=0;j<3;j++){
        if (j==0) {
            continue label2;
        }
        console.log("j="+j);
    }
    console.log("=====");
}
/*
打印结果
i=0
i=1
i=2
*/
```

需要注意，上面的代码没有打印出任何j的值。原因是当判断j等于0时，直接跳出外层循环的本次循环，因此内层循环的循环变量j并没有执行递增操作，j始终为0。

除了break和continue外，return语句也是一种中断语句，其用来为函数提供返回值。

3-8　异常抛出语句 throw

无论你是一个经验多么丰富的开发者，请相信我，在运行你所编写的程序时，错误一定会发生。在引擎执行ECMAScript代码时会发生各种各样的错误，可能是语法错误，可能是数据处理错误，也可能是由于服务端数据或用户输入的数据超出开发者的意料而发生的逻辑错误，等等。ECMAScript内置了许多异常对象，当某些行为触发了这些异常时，程序会将异常抛出并且中断在抛出异常的地方。开发者可以使用try-catch结构捕获并处理异常，从而

保证程序的顺畅执行，同样也可以使用throw语句来抛出一个异常。

暴露错误有时和解决错误一样重要。在开发中，你或许会编写大量有参数输入的函数，对于实际输入的参数，有时候并不是你可以决定的。例如，我们要编写一个标准的除法程序，代码如下：

```
function div(a,b){
    return a/b;
}
var res = div(3,4);
console.log(res);
```

上面的代码乍看起来很完美，实际上问题却很多，对于一个标准的除法运算，首先其除数与被除数应该都是标准的数值类型，不可以是字符串或者其他对象。其次，除数与被除数都不可以是无穷值，被除数也不可以为0。上面这两条规则都是对传入参数的限制。你或许会想，我们并没有方法控制用户要输入的东西，没错，但是你可以在用户输入了错误数据时让函数抛出异常，使程序中断。

JavaScript中使用throw关键字来抛出异常，修改上面的代码如下：

```
function div(a,b){
    if ((typeof a) != "number"||(typeof b) != "number"){
        throw "must input a Number Value";
    }
    if (!isFinite(a)|| !isFinite(b)) {
        throw "must input a Number is Finity";
    }
    if (b===0) {
        throw "Dividend must not be 0";
    }
    return a/b;
}
var res = div(3,4);
console.log(res);
```

当输入了非数值类型的参数、数值为无穷的参数或者被除数为0时，程序会直接中断，并在控制台打印出异常信息。如此，便十分严格地对函数的输入参数进行了约束。

使用throw关键字进行异常抛出的基本语法结构为throw exp。其中，throw为系统关键字，exp是要抛出的异常表达式，其可以为任意类型。例如，上面的示例代码中，我们实际上抛出的是一个字符串值异常，也可以抛出数值、布尔值或者任意自定义对象。例如，我们可以为某种特定的错误类型创建一个专门定义的对象，示例如下：

```
var InputError = {
    NotNumberError : "must input a Number Value",
    FinityError : "must input a Number is Finity",
    DividendError:"Dividend must not be 0"
};
function div(a,b){
    if ((typeof a) != "number" || (typeof b) != "number"){
        throw InputError.NotNumberError;
    }
    if (!isFinite(a)|| !isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b===0) {
        throw InputError.DividendError;
    }
    return a/b;
}
var res = div(3,4);
console.log(res);
```

通过自定义异常对象的方式更有利于结构化和标准化地对异常进行捕获与处理。现在，你已经学会了如何在代码中抛出异常，但仅仅抛出异常是远远不够的，后面我们将学习如何捕获异常、处理异常与传递异常。

3-9　对异常进行捕获处理

ECMAScript代码中抛出的异常如果不进行处理，程序就会直接中断，通常这并不是我们想看到的结果。ECMAScript中的try-catch-finally结构用来捕获和处理异常。依然使用上一个示例中所编写的除法器来做例子，在调用除法器函数时，传入一个错误的参数如下：

```
var InputError = {
    NotNumberError : "must input a Number Value",
    FinityError : "must input a Number is Finity",
    DividendError:"Dividend must not be 0"
};
function div(a,b){
    if ((typeof a) != "number" || (typeof b) != "number"){
        throw InputError.NotNumberError;
```

```
    }
    if (!isFinite(a)||!isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b===0) {
        throw InputError.DividendError;
    }
    return a/b;
}
var res = div("3",4);
console.log(res);
```

不出所料，程序运行时会抛出异常并直接中断。在实际开发中，遇到异常后，让程序中断并不是一个友好的选择（尽管用户输入的数据可能有些随心所欲），我们应该想办法将错误的信息提示给用户，并且让程序跳过这个异常。很多情况下，一个复杂的程序不只有一个功能，因为某个功能的异常而退出整个程序是让人无法接受的。try-catch-finally结构在ECMAScript中专门来捕获与处理异常。

多说一点，很多高级语言都有异常处理机制，有编程基础的同学一定对try-catch-finally十分熟悉，在Java语言中也有这样的结构。Objective-C语言中对应的结构为@try-@catch-@finally。Swift语言中的异常处理略微复杂一些，但是也有do-catch结构。

将上面的示例程序修改如下：

```
var InputError = {
    NotNumberError : "must input a Number Value",
    FinityError : "must input a Number is Finity",
    DividendError:"Dividend must not be 0"
};
function div(a,b){
    if ((typeof a) != "number" || (typeof b) != "number"){
        throw InputError.NotNumberError;
    }
    if (!isFinite(a)||!isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b===0) {
        throw InputError.DividendError;
    }
    return a/b;
}
```

```
try{
    var res = div("3",4);
}catch(e){
    console.log(e);
}finally{
    console.log("异常处理结束");
}
console.log(res);        //将打印 undefined
```

运行代码，控制台的打印结果如图3-1所示。

```
must input a Number Value
异常处理结束
undefined
[Finished in 0.1s]
```

图 3-1　进行异常捕获

可以发现，这次程序完整运行，下面我们来解释一下控制台的打印信息是如何来的。首先，try-catch-finally结构中的3个关键字里，try对应的代码块用来执行可能会抛出异常的代码，实际上，如果没有异常抛出，catch结构就是透明的，如果try关键字对应的代码块中有异常抛出，程序首先会进入catch代码块，catch代码块的作用是对抛出的异常进行捕获，其会将抛出的异常作为参数传入，我们拿到异常后，可以根据异常类型做相关逻辑处理。异常处理结束后，如果有finally代码块，会执行finally代码块中的代码，最后结束异常捕获处理结构。在try-catch-finally结构中，不一定3个代码块都要存在，可以有如下3种结构：

（1）try-catch

（2）try-finally

（3）try-catch-finally

需要注意，finally代码块并不对异常进行捕获，无论有没有异常抛出，finally代码块都会被执行。同样，如果没有catch代码块但是抛出了异常，尽管finally代码块存在，程序依然会中断。

3-10　传递异常

通过前面的示例，你已经学会了如何捕获与处理异常，下面我们将除法器函数再进行一层封装，将异常处理逻辑封装进新的函数内部，这样在其他开发者调用这个新的除法器函数时，就无须再担心程序中断，示例代码如下：

```
var InputError = {
    NotNumberError: "must input a Number Value",
    FinityError: "must input a Number is Finity",
    DividendError: "Dividend must not be 0"
};
function div(a, b) {
    if ((typeof a) != "number" || (typeof b) != "number") {
        throw InputError.NotNumberError;
    }
    if (!isFinite(a) || !isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b === 0) {
        throw InputError.DividendError;
    }
    return a / b;
}
function newDiv(a,b) {
    try {
        var res = div(a, b);
    } catch (e) {
        console.log(e);
    } finally {
        console.log("异常处理结束");
    }
    return res;
}
var res = newDiv("3",4);
console.log(res);
```

经过优化后，再调用上面代码中的newDiv()函数时，无论传入什么参数，都不会再抛出异常，异常已经在newDiv()函数内部被捕获并且处理了。如果我们不在newDiv()函数内部对异常进行捕获，会怎么样处理呢？我想你已经想到了，异常会被继续向上传递，从newDiv()函数抛出，示例如下：

```
function newDiv(a,b) {
    try {
        var res = div(a, b);
    } finally {
        console.log("异常处理结束");
```

```
        }
        return res;
    }
    try{
        var res = newDiv("3",4);
    }catch(e){
        console.log("newDiv 处理异常");
    }
```

控制台打印信息如图3-2所示。

```
异常处理结束
newDiv处理异常
undefined
[Finished in 0.1s]
```

图 3-2　异常的传递

在ECMAScript中，一旦某个函数抛出异常，异常首先会传递到此函数的调用方，如果调用方有对异常进行捕获处理，异常就不再向上传递，如果调用方没有对抛出的异常进行捕获处理，则此异常会继续向上传递，按照这样的逻辑，一层一层地向上传递，直到被捕获处理或者不再有可以处理的调用方。

明白了异常的传递机制，我们可以更加灵活地控制异常的传递，将异常进行分层处理，示例代码如下：

```
    function newDiv(a,b) {
        try {
            var res = div(a, b);
        } catch(e){
            if (e===InputError.DividendError) {
                res = NaN;
            }else if (e==InputError.FinityError) {
                res = NaN;
            }else{
                throw e;
            }
        }finally {
            console.log("内层异常处理结束");
        }
        return res;
    }
    try{
        var res = newDiv(Infinity,1);
    }catch(e){
        console.log("输入错误!");
    }finally{
```

```
        console.log("外层异常处理结束");
    }
    console.log(res);
```

上面的代码中，newDiv()函数只对被除数为零和参数有无穷值的异常进行了处理，将NaN直接作为结果返回。如果是参数类型错误，newDiv()函数就没有做任何逻辑代码，再次将异常抛出，交给调用方处理。

3-11 使用函数语句定义函数

几乎所有编程语言中都有函数这个概念，从功能上讲，函数是一组可以随时运行的代码语句，是一个代码块，也是一段子程序。ECMAScript是一种面向对象语言，在ECMAScript的世界中，万事万物都可以是对象。有趣的是，函数在ECMAScript中实质上也是一种功能完整的对象。ECMAScript中有3种基本的方式来定义函数，分别为函数语句定义法、函数表达式定义法和Function构造函数对象法。

ECMAScript中有一种特殊的语法可以直接来定义函数，在前面的示例中，我们也经常使用这种方法来定义函数，示例如下：

```
function outputName(){
    console.log("Jaki");
}
```

函数语句的语法结构可以简化为：function name(param...){}。其中，function为定义函数的关键字，name为函数的名称，通过函数名称可以直接调用函数，后面的小括号中可以定义函数的形参，大括号中为函数核心函数体，其中可以编写函数的具体逻辑代码。

需要注意，形参的全称为“形式参数”，其在定义函数的时候被定义，用来接收传入函数的参数。形参在整个函数内部使用，脱离函数内部，形参将没有任何意义。与形参对应的还有实参，实参全称为“实际参数”，是函数在调用时确确实实传递给函数的参数。一般情况下，在调用函数的时候，实参传递给形参，并且一一对应，但是ECMAScript语言中，允许开发者传入的实参与形参并不对应。

下面我们来定义一个带参数和返回值的函数，代码如下：

```
function outputHello(name){
    return "Hello "+name;
}
console.log(outputHello("Lucy"));        //打印出 Hello Lucy
```

上面的函数需要传入一个姓名作为参数，将姓名拼接上“Hello ”字符串后返回。在ECMAScript中，函数的返回值不需要专门定义（和其他主流编程语言有所区别），直接使用return关键字来返回即可。

事实上，在ECMAScript中，任何一个函数都有返回值，如果不显式地使用return关键字进行返回，或者只使用了return关键字，而没有返回任何值，函数实际的返回值为undefined。

在ECMAScript语言中，通过函数语句语法定义的函数，函数的调用和函数的定义并没有严格的顺序，其实我们也可以先调用某个函数，再进行此函数的定义，示例代码如下：

```
//先调用
console.log(outputHello("Lucy"));        //打印出 Hello Lucy
//后定义
function outputHello(name){
        return "Hello "+name;
}
```

上面的代码可以顺利执行的原因很简单，ECMAScript语言中有“变量提升”的概念。在程序执行之前，解释器会对全局声明的变量、定义的函数等进行预处理，因此在代码层面，函数的调用是在定义前还是在定义后都无关紧要。

3-12　使用函数表达式定义函数

在前面的示例中说过，函数是一种功能完整的对象，在ECMAScript中也可以使用函数表达式来定义函数对象。函数表达式看上去与函数语句十分相似，示例代码如下：

```
//函数表达式
var outputAge = function(age){
        console.log("My age is "+age);
};
outputAge(25);          //将打印 My age is 25
```

上面的代码在定义函数时，等号左边是一个变量，等号右边是一个函数表达式。函数表达式实质上返回了一个函数对象，将其赋值给outputAge变量，之后通过outputAge变量便可以直接对其所指向的函数进行调用。需要注意，函数表达式与函数语句的语法最大的区别在于其可以省略函数名，也可以理解为函数表达式创建了一个匿名函数。

函数表达式也可以创建有名称的函数，这在编写递归函数时十分重要，可以方便函数进行自调用，示例如下：

```
//定义一个递归阶乘函数
var mathFunc = function mathF(a){
    var res = a;
    a--;
    if (a>0) {
        res *= mathFunc(a);
    }
    return res;
};
var mathRes = mathFunc(5);
console.log(mathRes);       //120
```

需要注意，函数表达式定义的函数名只能在函数内部使用，对外部来说，实质上还是一种匿名函数。

还有一点需要特别注意，函数表达式定义的函数在函数定义之前是不能够进行调用的，你或许会觉得奇怪，程序解释器不是会对一些全局定义预处理呢？没错，但是函数表达式的实质是将一个函数对象赋值给一个变量，解释器只是预先定义好了变量符号，在表达式未执行之前，此变量实际是undefined，函数并不存在。这也是函数表达式与函数语句的另一重大区别，函数语句定义后的函数名是无法修改的，而函数表达式定义的函数只是将函数对象赋值给变量，变量可以重新赋值，也可以将函数传递给另一个变量，示例如下：

```
//函数表达式
var outputAge = function(age){
    console.log("My age is "+age);
};
//将 outputAge 函数传递给新的变量
var newFunc = outputAge;
//重新对 outputAge 赋值
outputAge = "Hello world";
newFunc(25);                //将打印 My age is 25
console.log(outputAge);     //将打印 Hello world
```

3-13　使用 Function 构造器定义函数

函数在ECMAScript中是功能完整的对象，关于对象，你现在了解的可能并不深入，不用担心，后面会有专门的章节为你介绍ECMAScript中的对象。现在你只需要简单知道：对象是属性和方法的包装。在ECMAScript中，也可以通过Function对象和new关键字来构造函

数对象，简单代码示例如下：

```
var output = new Function("name","console.log(name);");
output("Jaki");          //将打印输出 Jaki
```

Function构造函数的结构可以简化为：Function(param,param…,funcbody)。Function构造函数中的参数个数并不固定，最后一个参数为创建函数的函数体，前面所有的参数都将作为函数的形参。需要注意，所有的参数类型都需要为字符串类型，函数体也可以包装成字符串。

事实上，无论是通过哪种方式创建的函数，其实质都是Function类型的对象。Function实例对象中有一些内置的属性，常用的有arguments属性与length属性。arguments属性将返回一个数组结构数据，数组中是开发者传入的所有实参。length属性将返回函数形参的个数。有了arguments属性，我们在使用函数的时候就不必从形参获取外界传递至函数内部的数据了，示例如下：

```
function myFunc(){
    //将传入的参数倒叙输出
    for (var i = arguments.length - 1; i >= 0; i--) {
        console.log(arguments[i]);
    }
}
myFunc(1,2,3);          //将输出 3，2，1
```

需要注意，函数对象的arguments属性内部也有一个length属性，这是数组对象内置的属性，用于获取数组中元素的个数，并不是函数对象的length属性。

3-14　立即执行函数

关于ECMAScript中的函数，还有一种十分重要的用法是“立即执行函数”。无论是使用Function构造函数还是使用函数表达式定义函数，实际上都是创建了一个函数对象，将其赋值给一个变量。对于一些只需要在创建时执行一次就不再需要的函数，也可以使用如下方式编写：

```
(function(){
    console.log("run self");
})();        //将打印 run self
```

上面这种方式在实际开发中应用得十分广泛，可以用来包装作用域隐藏某些内部变量和方法。

3-15　编程练习

练习1：编写函数，要求使用while语句实现一个数的阶乘。

解析：

```
function func(n){
        while(n>1){
                return n*func(n-1);
        }
        return 1;
}
console.log(func(5));           //120
```

练习2：编写函数，要求输入一个整数，将其倒序输出。

解析：

```
function func(n){
        let res = "" ;
        let temp = n ;
        while(temp>=10){
                res =      res + Math.floor(temp%10);
                console.log(temp);
                temp = Math.floor(temp/10);
        }
        return    res+temp;
}
console.log(func(10142));           //24101
```

练习3：编写函数，将数组进行逆序并放回原数组中。

解析：

```
function func(array){
        var newArray = [];
        for (var i = array.length - 1; i >= 0; i--) {
                newArray.push(array[i]);
        }
        for (var i = newArray.length - 1; i >= 0; i--) {
                array[i] = newArray[i];
```

```
        }
}
var array = new Array(1,2,3,4,5);
func(array);
console.log(array);             //[ 5, 4, 3, 2, 1 ]
```

练习4：编写函数，输入一句话，输出最后一个单词的长度。

解析：

```
function func(string){
        let c = 0;
        for (var i = string.length-1; i >=0; i--) {
                if (string.charAt(i) == " ") {
                        if (c!=0) {
                                return c;
                        }
                }else{
                        c++;
                }
        }
        return c;
}

console.log(func('hello world'));          //5
```

字符串的charAt函数用来获取字符串中某个位置的字符。

练习5：有一对兔子，从出生后第3个月起，每个月都生一对兔子，小兔子长到第三个月后，每个月又生一对兔子，假如兔子都不死，问每个月的兔子总数为多少？

解析：

```
function func(m){
        if (m<=2) {
                return 1;
        }else{
                return func(m-1)+func(m-2);
        }
}
console.log(func(8));          //21
```

每个月兔子的数量实际上是一个斐波那契数列，即满足规律：1，1，2，3，5，8，13，

21……。从第3个月开始，兔子的数量都是前两个月兔子数量的和，因此使用递归来解决这个问题，十分简单。

练习6：打印出100～1000之间的所有“水仙花数”。所谓“水仙花数”，是指一个三位数，其各位数字的立方和等于该数本身。例如，153是一个“水仙花数”，因为$153=1^3+5^3+3^3$。

解析：

```
function func(){
    for (var i = 101; i <1000; i++) {
        let a = Math.floor(i/100);
        let b = Math.floor(i/10)%10;
        let c = i%10;
        if (i==a*a*a+b*b*b+c*c*c) {
            console.log(i);
        }
    }
}
func();            //153 370 371 407
```

练习7：利用条件运算符的嵌套来完成此题：学习成绩>=90分的同学用A表示，60～89分之间的用B表示，60分以下的用C表示。

解析：

```
function func(s){
    return s>=90?'A':(s>=60?'B':'C');
}
console.log(func(90));        //A
```

练习8：编写函数，输入字符串，分别统计其中字母、数字和其他字符的个数，并返回。

解析：

```
function func(string){
    let char = 0;
    let fig = 0;
    let o = 0;
    for (var i = 0; i < string.length; i++) {
        if(string.charCodeAt(i)>='A'.charCodeAt(0)&&string.charCodeAt(i)<='Z'.charCodeAt(0)){
            char++;
        }else if(string.charCodeAt(i)>='a'.charCodeAt(0)&&string.charCodeAt(i)<=
'z'.charCodeAt(0)){
            char++;
```

```
                    }else if(string.charCodeAt(i)>='0'.charCodeAt(0)&&string.charCodeAt(i)<=
'9'.charCodeAt(0)){
                        fig++;
                    }else{
                        o++;
                    }
                }
                return {char:char,fig:fig,o:o};
        }
        console.log(func("21dsd^%fs dsa"));            //{ char: 8, fig: 2, o: 3 }
```

练习9：试着编写函数解决问题：一球从100米高度自由落下，每次落地后反跳回原高度的一半；再落下，求它在第10次落地时，共经过多少米？第10次反弹多高？

```
        function func(){
                let h = 100;
                let sn = h;
                let hn = sn/2;
                for (var i = 2; i <= 10; i++) {
                        sn = sn+2*hn;
                        hn = hn/2;
                }
                return {sn:sn,hn:hn};
        }
        console.log(func());
```

练习10：猴子吃桃问题：猴子第一天摘下若干个桃子，当时吃了一半，还不过瘾，又多吃了一个，第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天早上想再吃时，见只剩下一个桃子了。求第一天共摘了多少。

```
        function func(){
                let day = 9;
                let a = 0;
                let b = 1;
                while(day>0){
                        a=(b+1)*2;
                        b = a;
                        day--;
                }
```

```
        return b;
}
console.log(func());            //1534
```

练习11：编程求解1+2!+3!+...+20!的和。

解析：

```
function func(){
        let s=0; t=1;
        for (var i = 1; i <=20; i++) {
                t = t*i;
                s = s+t;
        }
        return s;
}
console.log(func());            //2561327494111820300
```

第 4 章

ECMAScript 面向对象编程

相信你对“对象”一词并不陌生，前面的章节中我们多次提到对象这个概念。在代码的世界里，对象就是用来描述某个特定的事物，它是对现实世界中事物的抽象。面向对象是一种软件开发方式，在代码中使用对象来描述现实中的事物，可以使代码更易理解维护，同时抽象性、封装性和可重用性都会大大提高。从描述问题的方式来看，编程语言可以简单地分为面向过程语言和面向对象语言。面向过程语言以“数据结构+算法”的模式来解决问题。面向对象语言使用“对象+消息”的模式来解决问题。相比之下，面向过程语言编写的代码更简洁，运行效率更快，适合进行科学计算。

面向对象编程是一种计算机编程架构技术，简称OOP（Object Oriented Programming）。面向对象编程的基本原则是程序是由独立作用的对象组成的，面向对象编写的程序更抽象、更易懂，适合解决现实生活中的问题，使用面向对象编程技术开发的软件有着更好的重用性、灵活性和扩展性。本章首先介绍对象的相关知识，然后介绍ECMAScript中的面向对象编程方法以及实现面向对象编程的原理和手段。

4-1 创建对象

ECMAScript是一种基于对象的脚本语言，我们不仅可以使用语言内置的许多对象，也可以创建自己所需要的对象。使用字面量创建一个对象的语法非常简单，只需要使用大括号将需要封装的属性和方法进行包裹即可，示例如下：

```
var teacher = {};
console.log(typeof teacher);          //将打印 object
```

上面的第一句代码创建了一个空对象，将其赋值给了变量teacher。如果使用typeof运算符对teacher变量进行类型检查，你会发现它实际上是object类型，即对象类型。一般我们创建对象是为了描述某个事物，以teacher对象为例，我们想让它描述一个教师的个人信息和行为，一个空的对象是没有实际用途的，为teacher对象添加一些属性和行为，示例如下：

```
var teacher = {
    name:"珲少",
    age:25,
    subject:"JavaScript",
    teaching:function(){
        console.log("开始教学");
    },
    relaxing:function(){
        console.log("开始讲故事");
    }
};
```

上面的代码为teacher对象添加了3个属性和2个行为。name属性用来描述教师的姓名，age属性用来描述教师的年龄，subject属性用来描述教师的教学科目。teaching行为用来描述教师的教学行为，relaxing行为用来描述教师的放松行为。你应该明白了，属性描述的都是一些对象信息，其语法很像键值对，冒号左边是属性名，冒号右边是属性的值。行为描述的都是对象动作，在语法上冒号左边是行为名称，冒号右边是行为函数。需要注意，对象属性的值可以是任何类型，当然也可以是另一个对象。

使用“点语法”可以方便访问对象的属性和调用对象的方法。示例如下：

```
console.log(teacher.name);          //打印教师的姓名
console.log(teacher.age);           //打印教师的年龄
console.log(teacher.subject);       //打印教师的专业
```

```
teacher.teaching();            //执行教学动作
teacher.relaxing();            //执行休息动作
```

除了点语法外，JavaScript中还可以通过中括号法来访问对象属性与调用对象的方法，示例如下：

```
console.log(teacher["name"]);           //打印教师的姓名
console.log(teacher["age"]);            //打印教师的年龄
console.log(teacher["subject"]);        //打印教师的专业
teacher["teaching"]();                  //执行教学动作
teacher["relaxing"]();                  //执行休息动作
```

需要注意，使用中括号法访问的时候，中括号中的值必须为字符串类型。

4-2　设置对象的属性和行为

上一示例中在创建teacher对象时就已经为这个对象添加了属性和方法。很多时候，对象并不是一成不变的，现实生活中也是这样，比如几个月前笔者还在教Swift编程语言，现在已经和你交流JavaScript了。其实，我们可以随时为已经存在的对象添加新的属性方法或者修改某个属性方法。为对象追加或修改属性方法的语法也十分简单，示例如下：

```
//修改对象的属性和方法
teacher.subject = "Swift";          //将专业修改为 Swift
teacher.teaching = function(){
    console.log("Teaching Swift");
};
teacher.students = ["July","Jeke","Even"];          //添加一个学员列表
console.log(teacher);
```

同样，使用方括号法也可以完成对象的设置，示例如下：

```
//修改对象
teacher["subject"] = "Swift";       //将专业修改为 Swift
teacher["teaching"] = function(){
    console.log("Teaching Swift");
};
teacher["students"] = ["July","Jeke","Even"];       //添加一个学员列表
console.log(teacher);
```

你一定还记得，我们在学习运算符的时候提到过delete运算符，它的作用就是删除对象

的某个属性或方法。

在阅读JavaScript代码时，你可能会发现一个出现频率非常高的关键字：this。在对象内部行为使用的this关键字将指向当前对象本身（这是一般情况，并不是必然情况，this的指向会随着运行环境或调用者而灵活修改），例如：

```
var person = {
    name:"Jaki",
    sayHi:function(){
        console.log("Hi,My name is " + this.name);
    }
};
person.sayHi();        //将输出 Hi,My name is Jaki
```

上面代码中的this指的就是person对象本身，其实上面的代码和下面的代码功能完全一致：

```
var person = {
    name:"Jaki",
    sayHi:function(){
        console.log("Hi,My name is " + person.name);
    }
};
```

你可能会问，那我们直接使用person不就好了？为什么非要使用this呢？的确如此，针对上面的情况，我们没有必要使用this关键字。等到后面更加深入地学习了面向对象编程，尤其是动态生成对象的方法后，你就能真正体会到this关键字的强大之处了。现在就让我们拭目以待吧！

4-3　内置 Number 对象

在ECMAScript中有5种原始类型：Undefined、Null、Boolean、Number、String。针对Boolean、Number和String原始类型，在ECMAScript中还有其对象形式的包装，并且在必要时，ECMAScript会自动地在原始值和对象之间转换。由于这些内置对象的存在，我们可以方便地对数值、字符串等数据进行操作和处理。所谓内置对象，是针对自建对象而言的，在上一示例中，我们创建的“教师”对象就是自建对象。内置对象则是JavaScript中预先定义为开发者提供的对象，开发者可以直接使用。

ECMAScript中的Number对象是对原始类型Number的包装。可以使用Number构造函数来进行实例创建，首先需要传入被创建对象的数字值。示例如下：

```
var num1 = new Number(5);
console.log(num1);            //将打印[Number: 5]
console.log(typeof num1);          //将打印 object
```

如果传入Number方法中的参数无法转换成数字，就会返回NaN。需要注意，只有使用new关键字调用的方法才叫构造方法，才能构建出对象。若直接使用Number函数，则实际上会返回一个数值，这种方式通常用来强转值的类型。

在ECMAScript中，构造函数也是一种对象。实际上，Number函数本身也是一个对象（我们先将其称为Number函数对象），只是它的作用是生成其他对象（姑且把它生成的对象都叫作Number实例对象）。ECMAScript通过new关键字生成对象的原理这里先不做过多介绍，后面会专门讨论ECMAScript中的原型机制。现在，你只需要区别Number函数对象与Number实例对象即可。在Number函数对象中定义了许多有意义的常量属性，示例如下：

```
//可以表示两个数值之间的最小间隔
console.log(Number.EPSILON);                  //2.220446049250313e-16
//JavaScript 中最大的安全整数
console.log(Number.MAX_SAFE_INTEGER);         //9007199254740991
//能表示的最大数
console.log(Number.MAX_VALUE);                //1.7976931348623157e+308
//能表示的最接近 0 的数
console.log(Number.MIN_VALUE);                //5e-324
//非数字值
console.log(Number.NaN);                      //NaN
//负无穷大
console.log(Number.NEGATIVE_INFINITY);        //-Infinity
//正无穷大
console.log(Number.POSITIVE_INFINITY);        //Infinity
```

正负无穷值当产生溢出行为时会被返回。Number函数对象中也定义了一些常用的方法，示例如下：

```
//判断传入的参数是否是 NaN
console.log(Number.isNaN(1));
//判断是否是有限数字
console.log(Number.isFinite(1));
//判断是否为整数，字符串将输出位 false
console.log(Number.isInteger("1"));
```

```
//判断是否为安全的整数
console.log(Number.isSafeInteger(1));
//将字符串转换为浮点值
console.log(Number.parseFloat("1.23"));
//将字符串转换为整数值
console.log(Number.parseInt("123.12"));
```

上面的属性和方法都是Number函数对象定义的。

Number实例对象中也有一些预定义的方法，所有的Number实例对象共享这些方法，示例如下：

```
var num2 = new Number(123);
//将数字转换成科学计数法，传入的参数为保留小数的位数
console.log(num2.toExponential(2));          //1.23e+2
//将数字转换成字符串，传入的参数为保留小数的位数
console.log(num2.toFixed(2));                //123.00
//将数字转换为指定有效数字长度的数字，传入的参数为有效数字的位数
console.log(num2.toPrecision(2));            //1.2+e2
//将数字转换成字符串，传入的参数设置进制
console.log(num2.toString(10));              //123
//返回 Number 实例对象的原始值
console.log(num2.valueOf());
```

在使用toString()函数将数值对象转换为字符串时，可以传入参数来设置要转换成的进制。这个进制参数可以设置为一个2 ~ 36之间的整数，这个范围很好理解，数字0 ~ 9加上26个英文字母，最多可以表达36个数字，因此最大可以描述到36进制。

4-4　Number 对象与 Number 数值

ECMAScript是一门十分易用并且对异常要求不严格的语言。前面也提到，在必要的时候，无须开发者关心，ECMAScript会自动完成对象与原始值之间的转换。如果使用字面量创建了一个原始数值，实际上也可以调用上面的Number实例对象的方法，示例如下：

```
var num3 = 100;
console.log(typeof num3);          //number
var str = num3.toString();
console.log(typeof str);           //string
console.log(typeof num3);          //number
```

如上面的代码所示，需要注意，num3在调用toString()方法的时候，ECMAScript把其当作对象来处理，但实际上只是创建了一个中间对象，并没有真正将原始值转换成Number实例对象。ECMAScript的这种弱类型的设计风格使我们在编写代码时十分畅快。另外，再简单说一些new关键字，使用new关键字在构造对象时实际上执行了3个操作：首先创建一个空的对象建立原型链，之后执行构造函数，将函数中的this关键字与新建的对象进行绑定，最后将这个新建的对象返回。因此，new关键字最大的作用不仅是创建出一个新的对象，而是原型链的建立。在很多场景中，你可能会见到直接使用Number函数来创建数值，并不使用new关键字，需要注意，这种方式创建的是原始数值，并不是Number实例对象，通常我们可以使用这种方式来完成其他类型到数值类型的强转，示例如下：

```
var num4 = Number('5');
console.log(typeof num4);           //number
```

4-5　内置 String 对象

String对象是对字符串的一种包装。你一定还记得，在JavaScript中可以使用双引号或者单引号来创建字符串原始值，同样可以使用String()构造函数来创建字符串对象，示例如下：

```
var str1 = new String("Hello World");
console.log(str1);              //[String: 'Hello World']
```

你也可以不使用new关键字而直接使用String()函数，只是这样实际上是创建了一个原始类型的String字符串，并不是对象，这种方法常常用来进行字符串转换。在实际开发中，对字符串的操作将十分频繁。举例来说，进行URL协议解析时，开发者可能需要截取URL中定义的参数。在进行界面数据的填充时，可能需要对字符串进行拼接、插入或替换等操作。ECMAScript中的String对象中封装了许多属性和方法，帮助开发者便捷地对字符串进行操作。

任何String实例对象都包含一个length属性，通过length属性可以获取字符串长度，即字符个数，示例如下：

```
var str1 = new String("Hello World");
console.log(str1.length);           //11
```

ECMAScript是JavaScript的标准，JavaScript最初的设计是用于编写浏览器脚本，因此其String实例对象中内置了许多对HTML标签操作的方法，这不是本示例的重点，这里我们不再深入讨论，你只需要掌握字符串操作相关的方法即可，示例如下：

```
var str1 = new String("Hello World");
//返回特定位置的字符，下标从 0 开始
console.log(str1.charAt(0));            //H
//返回特定位置的字符编码值
console.log(str1.charCodeAt(0));        //72
//在字符串后进行拼接，将拼接后的字符串返回
console.log(str1.concat("!"));          //Hello World!
//获取某个字符在字符串中的索引，从前往后找，如果没有找到，将返回-1
console.log(str1.indexOf('l'));         //2
//获取某个字符在字符串中的索引，从后往前找，如果没有找到，将返回-1
console.log(str1.lastIndexOf('l'));     //9
//进行字符串的比较，若原字符串小于参数字符串，则返回小于 0 的数；大于则返回大于 0 的数；相等则返回 0
console.log(str1.localeCompare("Aello World"));          //1
//使用正则表达式对字符串进行匹配，匹配结果将返回一个对象
console.log(str1.match(/He/));          //[ 'He', index: 0, input: 'Hello World' ]
//使用正则表达式来匹配字符串，将匹配到的字符串进行替换
console.log(str1.replace(/He/,"AI"));          //AIllo World
//使用正则表达式来查找某个子串的位置，如果没有找到，就返回-1
console.log(str1.search(/He/));         //0
//截取范围内的子字符串
console.log(str1.slice(0,3));       //Hel
//分隔字符串，返回数组，其中第 1 个参数为进行分割的字符，第 2 个参数为返回最多的子串个数
console.log(str1.split("l",10));        //[ 'He', '', 'o Wor', 'd' ]
//进行字符串的截取，第 1 个参数为开始截取的位置，第 2 个参数为截取的长度
console.log(str1.substr(0,2));      //He
//截取下标间的子串
console.log(str1.substring(1,2));       //e
//将字符串转为小写
console.log(str1.toLocaleLowerCase());          //hello world
console.log(str1.toLowerCase());        //hello world
//将字符串转换为大写
console.log(str1.toLocaleUpperCase());          //HELLO WORLD
console.log(str1.toUpperCase());        //HELLO WORLD
//去掉字符串开头和结尾的空格
console.log(str1.trim());
//从字符串左侧去掉空格
console.log(str1.trimLeft());
```

```
//从字符串右侧去掉空格
console.log(str1.trimRight());
//获取字符串对象的原始值
console.log(str1.valueOf());
```

需要注意，上面的所有方法并没有修改原字符串对象，而是将结果以字符串原始值的形式进行返回。和大多数编程语言一样，ECMAScript中字符串的下标也是从0开始的。在上面的示例代码中，match、replace和search方法都是通过正则表达式来进行子串的匹配的，正则表达式用来描述一种匹配规则，后面我们会详细介绍。ECMAScript也会在必要的时候自动对String原始类型与String对象进行转换，这对开发者来说是透明的，你也可以直接使用原始字符串调用字符串对象的属性和方法，例如：

```
console.log("Hello".length);        //5
```

需要注意，对于字符串截取方法slice，若传入的参数为负数，则字符串长度减去这个负数的绝对值为最终参数，示例如下：

```
console.log("Hello World".slice(-4,-1));        //orl
```

4-6　与 HTML 相关的 String 方法

上一示例中介绍的都是String内置对象中封装的通用方法，还有一些方法是专门用来操作HTML文本的。例如，你可以直接调用这些方法将字符串包装成a标签或超级链接，示例如下：

```
var htmlString = "goTo";
console.log(htmlString.anchor("name"));            //<a name="name">goTo</a>
console.log(htmlString.big());            //<big>goTo</big>
console.log(htmlString.blink());            //<blink>goTo</blink>
console.log(htmlString.bold());            //<b>goTo</b>
console.log(htmlString.fixed());            //<tt>goTo</tt>
console.log(htmlString.fontcolor('red'));        //<font color="red">goTo</font>
console.log(htmlString.fontsize(5));            //<font size="5">goTo</font>
console.log(htmlString.italics());            //<i>goTo</i>
console.log(htmlString.link("https://www.xxx.com"));            //<a
href="https://www.xxx.com">goTo</a>
console.log(htmlString.small());            //<small>goTo</small>
console.log(htmlString.sub());            //<sub>goTo</sub>
console.log(htmlString.sup());            //<sup>goTo</sup>
```

4-7 内置 Boolean 对象

Boolean对象是对布尔类型原始值的一种包装，同样使用new关键字加构造函数的方法来创建，示例如下：

```
var bool = new Boolean(true);
console.log(bool);              //[Boolean: true]
```

Boolean无论是函数对象还是实例对象都没有封装什么实用的属性或方法。但是对于Boolean构造函数的使用，需要多加注意。Boolean构造函数中所传的参数不一定必须是原始布尔类型的值，若传入的参数为0、-0、null、false、NaN、undefined或者空字符串“”，则生成的Boolean对象的原始值为false；传入其他任何值，都将生成一个原始值为true的Boolean实例对象。示例如下：

```
//以下都将生成原始值为 false 的 Boolean 对象
console.log(new Boolean(0));
console.log(new Boolean(-0));
console.log(new Boolean(NaN));
console.log(new Boolean(undefined));
console.log(new Boolean(""));
console.log(new Boolean(false));
console.log(new Boolean(null));
//下面这些生成的是原始值为 true 的 Boolean 对象
console.log(new Boolean("false"));
console.log(new Boolean({}));
console.log(new Boolean(Infinity));
console.log(new Boolean(new Boolean(false)));
```

在使用Boolean实例对象时，有一点需要记住，对于JavaScript中的if条件语句，其判断条件如果是一个对象，就会被自动转换为true，因此无论Boolean实例对象的原始值是true还是false，在if条件判断中都将被作为true来处理。例如以下代码：

```
var bf = new Boolean(false);
if (bf) {
    console.log("执行了");
}
```

上面的示例依然会执行if结构中的代码。因此，要在判断条件中使用Boolean对象，需要取其原始值再进行条件判断，例如：

```
var bf = new Boolean(false);
if (bf.valueOf()) {
    console.log("执行了");
}
```

如果你只是想将其他某个类型的值转换为布尔类型，直接使用Boolean()函数即可，转换的规则和前面所说的一致，即0、-0、null、false、NaN、undefined或者空字符串“”将转换为布尔值false，其他都将转换为布尔值true。

4-8　内置 Array 对象

数组是一种非常常用的数据结构，在ECMAScript中，数组也是一种对象，只是它是列表形式的对象。简单理解，数组就是一种有序列表。例如，我们创建一个学生名单列表，代码如下：

```
var stus = ["Tom","Jaki","Lucy","Ami"];
console.log(typeof stus);           //object
```

上面的代码使用中括号进行数组的创建，这是一种非常便捷的创建数组的方法。当然，我们也可以使用Array构造方法来创建数组对象，这两种方式创建出的数组对象实质上是一样的，例如：

```
var array = new Array("Tom","Jaki","Lucy","Ami");
console.log(typeof array);          //object
```

数组中的元素类型并非要保持一致，它们可以是任意类型的组合。需要注意，数组中的元素是有序的，可以通过下标的方式对它们进行访问。数组的下标是从0开始的，示例如下：

```
//访问数组中第 1 个元素
console.log(array[0]);        //Tom
```

再回过头来看数组对象的构造方法，Array()这个方法中如果只传入1个参数且为数值类型，就会返回一个固定长度的空数组对象；若传入多个参数或非数值类型的参数，则会创建数组并将参数作为数组中的元素。

需要注意，虽然你也可以将数组理解为类似这样的对象：{0:"Tom",1: "Jaki"}，但访问数组中的元素并不能使用点语法，这样的写法将报错：array.0。原因是对象的属性名称若是

以数字开头的，则其为非法的属性名，不能和点语法结合使用，只能够通过中括号的形式访问。如果你创建一个自定义的对象，为其添加一个以数字开头的属性，依然无法使用点语句进行访问。你可能还会有一个疑问：以中括号方式进行属性的访问时，属性名不是必须是字符串格式的吗？确实如此，只是如果我们传入的是数值，ECMAScript就会自动帮我们处理为字符串格式。

要判断某一个值是否为数组类型，可以使用Array函数对象的isArray()方法，如果传入的参数是数组对象，就会返回true，否则将返回false，示例如下：

```
//判断某个值是否为数组
console.log(Array.isArray(array));          //true
```

数组实例对象中也封装了许多属性与方法，其中length属性可以用于获取数组中元素的个数：

```
var array = new Array("Tom","Jaki","Lucy","Ami");
console.log(array.length);              //4
```

需要注意，数组实例对象的length属性可以被任意修改，如果将其修改为小于数组元素个数，溢出的元素将会被清掉。因此，在修改数组实例对象的length属性时要格外注意，示例如下：

```
var array = new Array("Tom","Jaki","Lucy","Ami");
console.log(array.length);              //4
array.length = 2;
console.log(array);              //[ 'Tom', 'Jaki' ]
```

数组实例中封装的方法可以分为两类：一类会修改原数组对象，另一类不会修改原数组对象，会将操作结果以新的数组对象返回。下面这些方法都会对原数组对象进行修改：

```
var array = [0,1,2,3,4,5,6,7,8,9];
//删除数组最后一个元素
array.pop();
console.log(array);              //[ 0, 1, 2, 3, 4, 5, 6, 7, 8 ]
//在数组的末尾添加元素
array.push(9,10);
console.log(array);              //[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
//倒置数组
array.reverse();
console.log(array);              //[ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ]
//删除数组中的第 1 个元素
array.shift();
console.log(array);      //[ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ]
```

```
//对数组进行排序
array.sort(function(a,b){
    if (a>b) {
        return 1;
    }else if(a<b){
        return -1;
    }else{
        return 0;
    }
});
console.log(array);    //[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
//进行数组元素替换，第 1 个参数表示开始替换的元素下标，第 2 个参数表示要替换的元素个数，后面的表示要替换的元素列表
array.splice(6,4,"100","end");
console.log(array);          //[ 0, 1, 2, 3, 4, 5, ,'100','end' ]
//在数组开头进行元素的追加
array.unshift(0,0);
console.log(array);          //[ 0, 0, 0, 1, 2, 3, 4, 5, '100', 'end' ]
```

上面列出的方法中，除了sort()排序方法外，其他都很好理解。数组实例对象的sort()排序方法需要传入一个排序函数function(a,b)，这个函数有两个参数，是按照数组中的先后顺序进行比较的两个元素。若排序函数返回小于0的值，则表示a元素会排在b元素之前；若排序函数返回大于0的值，则表示a元素要排在b元素之后；若排序函数返回0，则a元素和b元素的相对位置不变。

下面这些方法不会修改原数组实例对象：

```
var array = [0,1,2,3,4,5];
//进行数组元素追加
console.log(array.concat(6,7,8,9));       //[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
//将所有元素以传入的参数分隔进行拼接
console.log(array.join("."));             //0.1.2.3.4.5
//截取子数组，第 1 个参数为开始截取的下标，第 2 个参数为停止截取的下标（不包含此位置）
console.log(array.slice(0,3));            //[ 0, 1, 2 ]
//将数组元素拼接成字符串，以逗号分隔
console.log(array.toString());            //0,1,2,3,4,5
//返回数组中指定元素的下标，从前往后找
console.log(array.indexOf(1));            //1
//返回数组中指定元素的下标，从后往前找
console.log(array.lastIndexOf(1));        //1
```

需要注意，数组中的元素是可以重复的。

在实际开发中，对数组最频繁的操作是进行数组的遍历。JavaScript中的数组实例对象支持很多遍历方法，例如：

```
var stus = ["Tom","Jaki","Lucy","Ami"];
//进行数组的逐个遍历，需要传入一个函数，此函数有 3 个参数，分别为遍历到的元素、该元素的下标和原数组。第 2 个参数将与遍历函数中使用的 this 关联
stus.forEach(function(element,index,array){
    console.log(element,index,array);
},stus);

//对数组进行检查，传入一个函数，函数中的参数分别为遍历到的元素、该元素下标和原数组。此函数需要返回一个布尔值，如果返回 ture，就继续遍历；如果返回 false，就停止遍历。当所有元素都遍历完成并且都返回 true 时，结果才为 true，否则都为 false
//every 方法还可以传入第 2 个参数，这个参数将与遍历函数中使用的 this 关联
var notHaveAmi = stus.every(function(element,index,array){
    console.log(this);
    console.log(element,index,array);
    if (element == "Ami") {
        return false;
    }else{
        return true;
    }
},stus);
console.log(notHaveAmi);        //false
//some 遍历方法与 every 对应，只是其回调如果全部返回 false，结果才为 false，否则为 true
var haveAmi = stus.some(function(element,index,array){
    console.log(this);
    console.log(element,index,array);
    if (element == "Ami") {
        return true;
    }else{
        return false;
    }
},stus);
console.log(haveAmi);       //true
//数组过滤器：当回调函数返回 true 时，代表此元素通过过滤，将其添加进新数组返回
var newArray = stus.filter(function(element,index,array){
    console.log(this);
```

```
        console.log(element,index,array);
        if (element == "Ami") {
            return true;
        }else{
            return false;
        }
    },stus);
    console.log(newArray);              //[ 'Ami' ]
    //map 方法将数组中的每一个元素执行回调，然后将返回值重新组成数组返回
    newArray = stus.map(function(element,index,array){
        return element+"!";
    },stus);
    console.log(newArray);              //[ 'Tom!', 'Jaki!', 'Lucy!', 'Ami!' ]
    //reduce 方法是数组累加器，其会按照数组从左向右的顺序依次调用回调函数，回调函数中有 3
个参数：第 1 个参数为上次执行累加回调的返回值，第 2 个参数为当前遍历的元素，第 3 个参数为其
下标值。reduce 方法的第 2 个参数可选，其为首次进行累加回调的初始值。如果不提供初始值，reduce
函数会从数组索引为 1 的位置开始，即跳过索引为 0 的元素
    var res = stus.reduce(function(acc,element,index){
        return acc+" "+element;
    },"Hello");
    console.log(res);       //Hello Tom Jaki Lucy Ami
    //与 reduce 方法类似，其从右向左开始遍历
    res  = stus.reduceRight(function(acc,element,index){
        return acc+" "+element;
    },"Hello");
    console.log(res);       //Hello Ami Lucy Jaki Tom
```

虽然在JavaScript中并没有严格要求数组的遍历过程不能对数组结构进行修改。但是建议在遍历的过程中不要进行增删元素的操作。若在遍历过程中向数组中新增元素，则新的元素不会被遍历到。若删除元素，则会造成意料之外的结果。

上面列举的数组实例对象遍历方法略微复杂，虽然有详尽的注释，但依然强烈建议一定要自己动手练习。实践出真知，熟练掌握这些遍历方法会让你以后的学习得心应手，加油！

4-9　内置 Date 对象

Date对象是ECMAScript中用来处理日期与时间的。同样，使用Date构造方法来进行Date实例对象的创建，在创建Date实例对象时有4种传参方式，示例如下：

```
//以当前时间创建 Date 对象
var date = new Date();
console.log(date);              //2017-02-27T02:11:03.945Z
//以指定时间戳创建 Date 对象，时间戳单位为毫秒
date = new Date(1483888888999);
console.log(date);              //2017-01-08T15:21:28.999Z
//以指定的字符串创建 Date 对象，字符串必须为标准格式的时间字符串
date = new Date("December 17, 2017 03:24:00");
console.log(date);              //2017-12-16T19:24:00.000Z
//设置年、月、日、时、分、秒、毫秒来创建 Date 对象
date = new Date(2017,10,11,8,10,40,133);
console.log(date);              //2017-11-11T00:10:40.133Z
```

若在构造函数中不传入任何参数，则默认使用当前日期时间创建Date实例对象。若传入一个数值类型的参数，则会将此数值作为时间戳来创建Date实例对象，时间戳的单位为毫秒，代表从1970年1月1日起经过的毫秒数。若传入的参数是一个字符串，则会以此字符串表示的日期时间创建Date对象，需要注意，传入的字符串参数必须为标准的日期时间格式字符串。如果传入的参数大于1个，就会按照年、月、日、时、分、秒、毫秒的顺序进行读参并创建Date对象。需要注意，若传入的参数在1个以上但是不足7个，则未传入的参数将以0代替，代表月数的整数值需是0 ~ 11之间的一个数值，0代表1月。

使用Date构造函数对象的now方法可以直接获取1970年1月1日至当前时刻的时间戳，单位为毫秒，示例如下：

```
//返回从 1970 年 1 月 1 日起至现在经过的毫秒数
console.log(Date.now());            //1488162786627
```

Date构造函数对象的parse方法用于解析一个日期时间字符串，将返回1970年1月1日至指定日期时间的时间戳，单位为毫秒，示例如下：

```
//解析一个日期时间字符串
console.log(Date.parse("December 17, 2017 03:24:00"));        //1513452240000
```

Date构造函数对象的UTC方法用于将指定的日期时间转换为时间戳，其参数规则与Date()构造函数中最长参数的形式一致，最多可以接收7个参数，示例如下：

```
//指定日期时间，返回时间戳
console.log(Date.UTC(2017,0,1,10,30,30,120));          //1483266630120
```

你也可以使用Date实例对象中封装的一些方法来直接获取想要的信息，示例如下：

```
var date = new Date();
//根据本地时间获取日期对象是当前月的第几天
```

```
console.log(date.getDate());
//根据本地时间获取星期，从 0 开始，0 表示周日
console.log(date.getDay());
//根据本地时间获取日期对象当前的年份
console.log(date.getFullYear());
//根据本地时间获取日期对象当前的小时，0~23
console.log(date.getHours());
//根据本地时间获取日期对象当前的分钟
console.log(date.getMinutes());
//根据本地时间获取日期对象当前的秒数
console.log(date.getSeconds());
//根据本地时间获取日期对象当前的毫秒数
console.log(date.getMilliseconds());
//根据本地时间获取日期对象当前的月份，从 0 开始，0 表示 1 月
console.log(date.getMonth());
//返回时间戳，单位毫秒
console.log(date.getTime());
//获取当前时区的时区偏移
console.log(date.getTimezoneOffset());
//根据通用时间获取当前日期对象是当前月的第几天
console.log(date.getUTCDate());
//根据通用时间获取当前日期对象的星期数，0 表示周日
console.log(date.getUTCDay());
//根据通用时间获取日期对象当前的年份
console.log(date.getUTCFullYear());
//根据通用时间获取日期对象当前的小时
console.log(date.getUTCHours());
//根据通用时间获取日期对象当前的分钟
console.log(date.getUTCMinutes());
//根据通用时间获取日期对象当前的秒数
console.log(date.getUTCSeconds());
//根据通用时间获取日期对象当前的毫秒数
console.log(date.getUTCMilliseconds());
//根据通用时间获取日期对象当前的月份，从 0 开始，0 表示 1 月
console.log(date.getUTCMonth());
```

UTC是协调世界时的简称，其又称为世界统一时间或世界标准时间，被应用于许多互联网标准中。

下面这些方法用来修改Date实例对象：

```
var date = new Date();
//根据本地时间为日期对象设置月份中的第几天
date.setDate(10);
//根据本地时间为日期对象设置年份
date.setFullYear(1999);
//根据本地时间为日期对象设置小时
date.setHours(11);
//根据本地时间为日期对象设置毫秒数
date.setMilliseconds(123);
//根据本地时间为日期对象设置分钟数
date.setMinutes(30);
//根据本地时间为日期对象设置月份
date.setMonth(1);
//根据本地时间为日期对象设置秒数
date.setSeconds(30);
//根据时间戳来设置日期对象的时间，如果早于 1970 年 1 月 1 日，可以设置为负值
date.setTime(1488167242644);
//根据通用时间为日期对象设置月份中的第几天
date.setUTCDate(10);
//根据通用时间为日期对象设置年份
date.setUTCFullYear(1970);
//根据通用时间为日期对象设置毫秒数
date.setUTCMilliseconds(123);
//根据通用时间为日期对象设置分钟数
date.setUTCMinutes(30);
//根据通用时间为日期对象设置月份
date.setUTCMonth(1);
//根据通用时间为日期对象设置秒数
date.setUTCSeconds(30);
```

下面这些方法用于对Date实例对象进行格式化：

```
var date = new Date();
//以一种易读的方式返回日期
console.log(date.toDateString());          //Mon Feb 27 2017  星期  月  日  年
//返回符合 ISO 标准的日期字符串
console.log(date.toISOString());           //2017-02-27T05:13:25.025Z
//使用 toISOString() 返回一个表示该日期的字符串
console.log(date.toJSON());                //2017-02-27T05:14:11.414Z
//返回一个表示该日期对象日期部分的字符串，该字符串格式与系统设置的地区关联
```

```
    console.log(date.toLocaleDateString());        //2/27/2017
    //返回一个表示该日期对象的字符串，该字符串与系统设置的地区关联
    console.log(date.toLocaleString());            //2/27/2017, 1:16:27 PM
    //返回一个表示该日期对象时间部分的字符串，该字符串格式与系统设置的地区关联
    console.log(date.toLocaleTimeString());        //1:17:00 PM
    //返回一个表示该日期对象的字符串
    console.log(date.toString());                  //Mon Feb 27 2017 13:17:48 GMT+0800 (CST)
    //以人类易读格式返回日期对象时间部分的字符串
    console.log(date.toTimeString());              //13:18:19 GMT+0800 (CST)
    //把一个日期对象转换为一个以 UTC 时区计时的字符串
    console.log(date.toUTCString());               //Mon, 27 Feb 2017 05:18:43 GMT
    //从 1970 年 1 月 1 日 0 时 0 分 0 秒（UTC，即协调世界时）到该日期的毫秒数，与 getTime()方
法一致
    console.log(date.valueOf());
```

需要注意，若在设置Date实例对象的年份时使用了两位数，则将默认表示为1900 ~ 1999年之间的年份，例如：

```
    var date = new Date(90,9,22);
    console.log(date);              //1990-10-21T16:00:00.000Z
```

为了避免不必要的歧义，最好不要使用两位数来设置Date实例对象的年份。

4-10　内置 Math 对象

在编程过程中，数学运算十分重要，毕竟计算机学科的基础就是数学。在各种编程语言中也都提供了数学函数库供开发者使用。在ECMAScript中，Math是一个内置对象而并非函数对象。这和我们之前学习的内置对象有些不同，这也很容易理解，Math对象的作用是提供便利的数学方法，我们并不需要创建出实例对象。

在开发中可能经常会用到一些数学常量，例如圆周率、自然对数等。Math对象中包装了一些属性可以直接获取这些常量，示例如下：

```
    //数学常量
    //自然常数
    console.log(Math.E);                //2.718281828459045
    //2 的自然对数
    console.log(Math.LN2);              //0.6931471805599453
    //10 的自然对数
```

```
console.log(Math.LN10);              //2.302585092994046
//以 2 为底 E 的对数
console.log(Math.LOG2E);             //1.4426950408889634
//以 10 为底 E 的对数
console.log(Math.LOG10E);            //0.4342944819032518
//圆周率
console.log(Math.PI);                //3.141592653589793
//1/2 的平方根
console.log(Math.SQRT1_2);           //0.7071067811865476
//2 的平方根
console.log(Math.SQRT2);             //1.4142135623730951
```

也可以使用Math对象中提供的方法来进行数学运算，常用方法列举如下：

```
//数学方法
//求绝对值
console.log(Math.abs(-4));           //4
//求反余弦值
console.log(Math.acos(0.5));         //1.0471975511965976
//求反正弦值
console.log(Math.asin(0.5));         //0.5235987755982988
//求反正切值
console.log(Math.atan(0.5));         //0.46364760900080615
//需要传入两个参数 x、y，求 x/y 的反正切值
console.log(Math.atan2(1,2));        //0.4636476090008061
//进行向上取整
console.log(Math.ceil(1.1));         //2
//求余弦值
console.log(Math.cos(0.5));          //0.8775825618903728
//传入参数 n，求自然对数 E 的 n 次方
console.log(Math.exp(2));            //7.38905609893065
//向下取整
console.log(Math.floor(2.9));        //2
//传入参数 n，求以自然常数 E 为底 n 的对数
console.log(Math.log(10));           //2.302585092994046
//求一组数中的最大值
console.log(Math.max(1,2,5,3,7));    //7
//求一组数中的最小值
console.log(Math.min(1,2,5,3,7));    //1
//传入两个参数 x、y，求 x 的 y 次方
```

```
console.log(Math.pow(2,3));              //8
//返回一个 0~1 之间的随机数
console.log(Math.random());
//进行四舍五入
console.log(Math.round(3.4));            //3
//求正弦值
console.log(Math.sin(0.5));
//求平方根
console.log(Math.sqrt(2));               //1.4142135623730951
//求正切值
console.log(Math.tan(1));
```

需要注意，上面所列举的三角函数方法的返回值都是以弧度为单位的。

4-11　内置 RegExp 正则表达式对象

关于正则表达式，我们前边在介绍String对象时提到过，正则表达式就是用来定义一种规则，通过规则来对文本进行匹配。在ECMAScript中，正则表达式也是一种对象，其可以使用RegExp构造函数来创建。

你可以通过两种方式创建正则表达式对象，最简单的方式是通过字面量来创建正则表达式，示例如下：

```
//通过字面量来创建正则表达式
var reg = /hello/i;
```

通过字面量创建正则表达式有这样的规则：斜杠符内编写正则表达式文本，结束斜杠的右侧指定匹配模式。匹配模式可以是表4-1所示的几种标志的组合。

表4-1　正则表达式的匹配模式

匹配模式标志	作　用
G	全局匹配（匹配到一个结果后，还会继续向后匹配，直到结束）
I	匹配过程忽略大小写
M	多行匹配模式（默认情况下，开始符^与结束符$是工作在单行模式的，将只匹配整个文本的开始与结束，配置这个参数后，其会匹配每一行的开始与结束）

例如，我们对“Hello world hello”进行不区分大小写的全局匹配，将会匹配到“Hello”和“hello”，示例如下：

```
var reg = /hello/ig;
var res = "Hello world hello".match(reg)
console.log(res);          //[ 'Hello', 'hello' ]
```

使用RegExp构造函数对象来创建正则表达式对象，示例如下：

```
var reg2 = new RegExp("hello",'ig');
console.log("Hello world hello".match(reg2));         //[ 'Hello', 'hello' ]
```

RegExp()构造函数中可以传入两个参数（第2个参数可选），第1个参数为正则表达式字符串，第2个参数为匹配模式。

RegExp构造方法中的第1个参数可以传入字符串形式的正则表达式，也可以直接传入字面量语法的正则表达式，例如如下语法也是正确的：

```
var reg2 = new RegExp(/hello/,'ig');
console.log("Hello world hello".match(reg2));         //[ 'Hello', 'hello' ]
```

但是需要额外注意，当RegExp构造方法中第1个参数为字符串时，如果此字符串中有特殊字符，就需要进行转义。例如，下面这两个正则对象是完全等价的：

```
var re = new RegExp("\\w+");
var re = /\w+/;
```

关于正则表达式，还有很多东西，我们可以深入探究一下。正则表达式又称规则表达式，其通过一些符号的组合来定义一种搜索算法。简单理解，一个正则表达式就是一个逻辑公式，通过公式来对文本进行精确或模糊搜索。正则表达式中定义了一些特殊字符，大致可以分为4类：字符类别、字符集合、边界、数量词。

字符类别用于模糊匹配，即某一个特殊字符可以代表某一类字符。常用的特殊字符列表如表4-2所示。

表4-2　常用的特殊字符

字　符	含　义	示　例
.（小数点）	匹配任意字符（换行符除外）	将匹配 hell： reg = /h..l/; "hello".match(reg)
\d	匹配 0～9 间的任何一个数字字符	将匹配 5h： reg = /\dh/; "5hello".match(reg);
\D	匹配任意一个不是数字的字符	将匹配 Eh： reg = /\Dh/; "Ehello".match(reg)

（续表）

字　符	含　义	示　例
\w	匹配一个字母、数字或下画线字符	将匹配 he： reg = /h\w/; "Ehello".match(reg);
\W	匹配任意非字母、非数字和非下画线的字符	将匹配 h$： reg = /h\W/; "h$llo".match(reg);
\s	匹配一个空白符，包括空格、制表符、换页符、换行符等	将匹配 h e： reg = /h\se/; "h ello".match(reg);
\S	匹配任意一个非空白字符	将匹配 he： reg = /h\S/; "hello".match(reg);
\t	匹配一个水平制表符	
\r	匹配一个回车符	
\n	匹配一个换行符	
\v	匹配一个垂直制表符	
\f	匹配一个换页符	
[\b]	匹配一个退格符	
\0	匹配一个 NUL 字符	
\xhh	匹配编码为 hh 的字符（十六进制）	
\uhhhh	匹配 Unicode 值为 hhhh 的字符	

字符集合用于匹配某个集合内的字符，具体含义的使用示例如表4-3所示。

表4-3　字符集合的含义与示例

字符集合	含　义	示　例
[abc]	匹配中括号内的一个字符	将匹配 he： reg = /h[abced]/; "hello".match(reg);
[a-b]	匹配范围内的字符	将匹配 he： reg = /h[a-e]/; "hello".match(reg);
[^abc]	匹配除了集合字符外的所有字符	
[^a-b]	匹配除了集合范围外的所有字符	

表4-4所示的特殊字符用来定义边界。

表4-4　用来定义边界的特殊字符

字　符	含　义	示　例
^	匹配字符串的开头	将匹配到 h： reg = /^h/; "hello".match(reg);
$	匹配字符串的结尾	将匹配到 o： reg = /o$/; "hello".match(reg);

表4-5中的特殊字符用来设置匹配字符的数量。

表4-5　匹配字符数量的特殊字符

字　符	含　义	示　例
x*	匹配 0 次或多次 x 字符	将匹配到 ell： reg = /el*/; "hello".match(reg);
x+	匹配 1 次或多次 x 字符	将匹配到 ell： reg = /el+/; "hello".match(reg);
x?	匹配 0 次或 1 次 x 字符	
x(?!y)	当 x 字符后面不是 y 字符时才匹配	将匹配到 lo： reg = /l(?!l)./; "hello".match(reg);
x|y	匹配字符 x 或者字符 y	
x{n}	匹配连续 n 个 x	将匹配到 ll： reg = /l{2}/; "hello".match(reg);
x{n,}	匹配至少连续 n 个 x	
x{n,m}	匹配连续出现 n～m 个 x	

上面我们使用很大篇幅来介绍有关正则表达式的基础知识，这是十分有必要的。正则表达式广泛应用于用户表单的验证。例如，验证用户输入的邮箱、手机号等是否合法。再回到RegExp实例对象，其中的一些属性可以用来获取设置的正则匹配模式，示例如下：

```
var reg = new RegExp(/l./);
//是否开启全局匹配模式
console.log(reg.global);           //false
//是否开启忽略大小写
console.log(reg.ignoreCase);       //false
```

```
//是否开启多行模式
console.log(reg.multiline);            //false
```

RegExp实例对象中定义了如下方法，可以直接对一个目标字符串进行检测：

```
var reg = new RegExp(/l./);
//对目标字符串进行正则匹配
console.log(reg.exec("hello"));            //[ 'll', index: 2, input: 'hello' ]
//检测目标字符串是否可以通过正则验证，即匹配到结果
console.log(reg.test("hello"));            //true
```

在实际开发中，RegExp实例对象的test方法要比exec方法更加常用，很多情况下，我们都需要用它来验证一个字符串是否符合规则。

4-12　内置 Function 对象

函数对你来说一定不陌生。我们知道，函数实际上也是一种对象，其是由Function构造函数创建出来的。Function实例对象中的arguments属性可以获取调用函数时所有传递进来的实参。length属性可以获取当前函数的形参个数。本示例中，你将学习到几个Function实例对象的方法，这些方法在ECMAScript面向对象技术中十分重要。

在ECMAScript中，this是一个十分重要的关键字，也是略微难于理解的关键字。在函数内部，this的值取决于函数是如何调用的，若直接调用全局函数，则函数内部的this指向全局对象。如果函数作为某个对象的行为被调用，那么其中的this指向该对象，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(){
        console.log("姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString();            //姓名：Jaki、年龄：25
```

由于toString方法是由teacher对象调用的，toString方法中的this就代表当前teacher对象本身。当在构造函数中使用this时，则情况不同，new关键字会创建一个空的对象，然后将构造函数中的this和这个对象进行绑定。

了解了this关键字，我们再来看Function实例对象中的几个方法就十分容易理解了。Function实例对象的apply方法用于指定调用方法的this值和参数。我们把前面的代码略作修改，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString("Teacher");        //Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
teacher.toString.apply(student,["Student"]);        //Student 姓名：Lucy、年龄：23
```

上面的示例代码中创建了两个对象：teacher与student。student对象并没有toString方法，但是我们可以借助teacher对象的toString方法来打印student对象的信息。Function实例对象中提供了apply方法，这个方法可以接收两个参数，第1个参数设置调用方法的上下文，即函数中this关键字的指向；第2个参数为一个数组，数组中的元素将被作为实参传入函数。有了apply方法，我们可以实现某个对象调用其他对象的方法。如上面的代码所示，就好像student对象借用了teacher对象的方法一样。

与apply方法十分类似的还有call方法，其作用也是设置所调用函数中this的指向与传入参数，不同的是call函数中的参数个数不定，第1个参数为要绑定到this的对象，之后的所有参数都会作为实参传入函数，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString("Teacher");        //Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
teacher.toString.call(student,"Studeng");        //Student 姓名：Lucy、年龄：23
```

Function实例对象中还提供了一个bind方法，这个方法将会创建一个新的函数，bind方法中第1个参数为新创建函数调用时的this指向，之后所有的参数都将作为默认内置实参传入函数，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString("Teacher");      //Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
var studentToString = teacher.toString.bind(student,"Student");
studentToString();           //Student 姓名：Lucy、年龄：23
```

需要注意，bind函数中定义的参数在传入新生成的函数时，会被置为内置参数，放在所有实参之前，且不会被覆盖掉，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
        console.log(arguments);
    }
}
teacher.toString("Teacher");      //Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
var studentToString = teacher.toString.bind(student,"Student");
//Student 姓名：Lucy、年龄：23
//{ '0': 'Student', '1': 'Hello' }
studentToString("Hello");
```

4-13　内置 Object 对象

在ECMAScript中，Object构造函数可以理解为一个对象包装器，使用这个构造函数可以创建出最原始的实例对象。我们知道，在ECMAScript中，自定义对象有两种方式，示例如下：

```
//使用字面量语法创建对象
var teacher = {
        name:"Jaki",
        age:25,
        teaching:function(){
                console.log("teching...");
        }
};
//使用 Object 构造函数创建对象
var student = new Object();
student.name = "Lucy";
student.age = 24;
student.learning = function(){
        console.log("learning...");
};
```

当我们使用Object()构造函数来进行对象的创建时，里面可以传入一个参数，构造函数会根据传入的参数类型创建相应的对象实例。

例如上面定义的对象都可以直接通过点语法或者中括号的方式进行属性和方法的访问。同样，我们也可以自由地对对象中的属性方法进行枚举、修改、删除等。实际上，对象中的属性和方法都有一套配置参数，这些配置参数决定了对象的可读性、可枚举性和可配置性等。

4-14　进行对象属性的配置

Object构造方法对象中提供了一个defineProperty()方法，这个方法会直接在一个对象上定义一个新的属性或者修改一个已经存在的属性。示例如下：

```
//使用 Object 构造函数创建对象
var student = new Object();
```

```
student.name = "Lucy";
student.age = 24;
student.learning = function(){
    console.log("learning...");
};
Object.defineProperty(student,"name",{
    configurable:true,
    enumerable:true,
    writable:true,
    value:"July"
});
console.log(student.name);          //July
```

上面的代码使用defineProperty()方法对student对象的name属性进行了重新配置，这个方法中需要传入3个参数，第1个参数为要进行配置的对象，第2个参数为需要进行配置的属性名，第3个参数为配置描述参数。

下面解释配置描述中可选的4个配置项所代表的意义。configurable用来设置此属性是否是可配置的，当第一次使用defineProperty方法对某个属性进行配置时，若将此配置项设置为false，则之后不可以再对该对象的这个属性的配置进行修改。enumerable配置此属性的可枚举性，若设置为true，则此属性可以被for-in结构遍历到，若配置为false，则此属性不能被for-in结构遍历到。writable配置此属性的可写性，若设置为true，则此属性可以被赋值运算符修改，若配置为false，则此属性不能被赋值运算符修改。value项其实就是设置当前属性的值。

你也可以在defineProperty()方法的描述参数中定义getter与setter方法。getter方法的返回值会被作为该属性的值，setter方法在属性被赋值时被调用，示例如下：

```
//使用 Object 构造函数创建对象
var student = new Object();
student.name = "Lucy";
student.age = 24;
student.learning = function(){
    console.log("learning...");
};
var name = "Lucy";
Object.defineProperty(student,"name",{
    configurable:true,
    enumerable:true,
    get:function(){
        console.log("正在使用 name 属性");
        return name;
```

```
        },
        set:function(value){
            console.log("将要设置 name 属性");
            name = value;
        }
});
console.log(student.name);          //正在使用 name 属性  Lucy
student.name = "July";              //将要设置 name 属性
console.log(student.name);          //正在使用 name 属性  July
```

需要注意，描述参数中的value、writable两个配置项与get、set两个配置项不能同时存在，否则JavaScript代码在运行时会抛出异常。有了属性的get和set配置，我们可以方便地监听对象某个属性的设置或获取，添加需要的业务逻辑。

Object构造函数对象中还提供了一个方法可以一次定义或修改多个属性，示例如下：

```
var teacher = {
        name:"Jaki",
        age:25,
        teaching:function(){
            console.log("teching...");
        }
};
Object.defineProperties(teacher,{
        "name":{
            value:"珲少",
            writable:false
        },
        "age":{
            value:25,
            writable:false
        }
});
```

4-15　Object 函数对象常用方法

除了前面示例提到的配置属性方法外，Object构造函数对象中还有一些其他针对实例对象操作的方法，其中assign方法可以实现将几个目标对象中可枚举的属性复制进源对象中，示例如下：

```
var teacher = {
    name:"jaki",
    age:24,
    teaching:function(){
        console.log("teaching");
    }
};
var teacher2 = {
    subject:"JavaScript"
}
Object.defineProperty(teacher,"number",{
    value:1001,
    enumerable:false
});
console.log(teacher.number);            //1001
for(prop in teacher){
    console.log(prop);
}
//进行对象可枚举属性的复制
var obj = {};
//第 1 个参数为目标对象，其后的参数为将被复制属性的对象
Object.assign(obj,teacher,teacher2);
console.log(obj.name+obj.age+obj.subject);        //jaki24JavaScript
obj.teaching();           //teaching
console.log(obj.number);          //undefined
```

上面的代码将teacher对象和teacher2对象中的可枚举属性全部复制进了obj对象中，number属性是不可枚举的，因此该属性并没有被复制。除此之外，从原型继承来的属性也不会被复制（原型与继承后面会具体介绍）。需要注意，在进行属性复制时，源对象不可以是null，若源对象中的属性名和目标对象的属性名重复，则源对象的此属性值会被覆盖。还有一点需要注意，assign方法进行的复制都是浅复制，即若目标对象的某个属性值是引用类型，则它复制的是此引用，并不是引用所对应的值，示例如下：

```
//深浅拷贝
var obj1 = {
    a:{
        name:"Jaki"
    },
    b:25
```

```
};
var obj2 = {};
Object.assign(obj2,obj1);
//修改 obj1
obj1.a.name = "Lucy";
obj1.b = 23;
//obj2 中的 b 并没有被修改，因为其是原始值类型，但是 a 属性被修改了，因为其是引用类型
console.log(obj2);            //{ a: { name: 'Lucy' }, b: 25 }
```

Object构造方法中的create函数也用于创建一个对象，其创建对象的时候可以指定对象的原型和若干属性。这里不需要深入理解对象的原型。简单地理解，原型是对象所继承的对象（类似于“父类”），对象可以直接使用原型中定义的属性。例如，对于所有讲JavaScript 教程的教师，我们可以定义一个父对象，其中定义一个所教课程的属性，其他由此对象继承而来的子对象中都可以使用这个属性，示例代码如下：

```
//继承的实现
var base = {
    subject:"JavaScript"
}
var teacher1 = Object.create(base,{
    "name":{
        value:"Jaki",
        enumerable:true
    },
    "age":{
        value:25,
        enumerable:true
    }
});
console.log(teacher1);        //{ name: 'Jaki', age: 25 }
console.log(teacher1.subject);            //JavaScript
```

需要注意，create方法中的两个参数，第1个参数为所创建对象的原型，第2个参数为要创建对象的属性配置列表，其和defineProperties方法中第2个参数的含义一致。

Object构造函数中的freeze函数用于冻结对象。冻结一词指的是不能向对象中添加新的属性，不能修改或者删除对象的属性，同样也不能对对象中属性的配置进行修改。简单理解，冻结的对象是一个完全不可变的对象。freeze方法传入一个对象作为参数，然后返回被冻结的对象。示例如下：

```
var fre = {
        name:"Jaki"
};
fre = Object.freeze(fre);
fre.name = "Lucy";
//冻结的对象不能修改
console.log(fre);         //{ name: 'Jaki' }
```

Object构造方法中的isFrozen方法可以获取某个对象是否为被冻结的对象。

Object构造函数对象中的getOwnPropertyDescriptor函数用来获取对象某个属性的配置信息，示例如下：

```
var fre = {
        name:"Jaki"
};
/*
{ value: 'Jaki',
   writable: true,
   enumerable: true,
   configurable: true }
*/
var des = Object.getOwnPropertyDescriptor(fre,"name");
```

Object构造函数对象中的getOwnPropertyNames函数可以获取指定对象所有自身的属性，从原型继承来的属性不包括在内。需要注意，这个方法会将可枚举和不可枚举的属性都获取到。示例如下：

```
var base = {
        subject:"JavaScript"
}
var teacher1 = Object.create(base,{
        "name":{
                value:"Jaki",
                enumerable:true
        },
        "age":{
                value:25,
                enumerable:true
        }
});
console.log(Object.getOwnPropertyNames(teacher1));          //[ 'name', 'age' ]
```

Object构造函数对象中的getPrototypeOf方法可以获取到某个对象的原型对象，示例如下：

```
var base = {
	subject:"JavaScript"
}
var teacher1 = Object.create(base,{
	"name":{
		value:"Jaki",
		enumerable:true
	},
	"age":{
		value:25,
		enumerable:true
	}
});
console.log(Object.getPrototypeOf(teacher1));		//{ subject: 'JavaScript' }
```

Object构造函数对象中的seal方法用来密封对象，密封对象不能添加新属性，不能删除已有属性，不能修改属性的配置，但是可以修改其属性的值。密封与冻结的唯一不同是对象属性的值是否可以修改。示例如下：

```
var seal = {
	name:"Jaki"
};
self = Object.seal(seal);
//密封对象不能添加新属性
seal.age = 25;		//undefined
console.log(seal.age);
```

isSealed方法用来判断一个对象是否是密封的。

除了冻结与密封外，对象还有“扩展”的概念。可扩展表示对象可以添加新的属性，不可扩展表示对象不能够添加新的属性，但是可以删除和修改已有的属性。Object构造方法中也有相应的函数对对象的可扩展性进行控制，preventExtensions方法用于抑制对象的扩展性，示例如下：

```
var ext = {
	name:"Jaki"
}
//抑制对象扩展
ext = Object.preventExtensions(ext);
```

```
ext.age = 25;
console.log(ext.age);          //undefined
```

同样，也有isExtensible方法来判断某个对象是否是可扩展的。

Object构造方法对象中还定义了一个keys方法，其会返回对象自身可枚举的属性名。需要注意，其和for-in遍历结构还是有一些区别的，keys方法只会返回对象自身的可枚举属性，而for-in遍历可以遍历出对象原型链上的可枚举属性。示例如下：

```
var base = {
    subject:"JavaScript"
}
var teacher1 = Object.create(base,{
    "name":{
        value:"Jaki",
        enumerable:true
    },
    "age":{
        value:25,
        enumerable:true
    }
});
console.log(Object.keys(teacher1));          //[ 'name', 'age' ]
```

直接打印对象实际上会打印出对象中可枚举的属性。

4-16　Object 实例对象常用方法

上一实例中介绍的是Object函数对象封装的一些常用方法。Object实例对象中也有一些方法，主要与其属性的检测相关。hasOwnProperty方法可以用来检查对象中是否包含某个属性，此属性必须是对象本身的，不能是从原型链上继承来的，示例如下：

```
var teacher = {
    name:"Jaki",
    age:25
}
//判断某个对象本身是否包含指定的属性，此属性不是原型链上的
console.log(teacher.hasOwnProperty("name"));          //true
```

isPrototypeOf方法用来检查当前对象是否在某个对象的原型链上，示例如下：

```
var teacher = new Object();
var prototype =   {
        subject:"JavaScript"
};
//设置原型
Object.setPrototypeOf(teacher,prototype);
teacher.name = "Jaki";
teacher.age= 25;
teacher.teaching = function(){
        console.log("teaching");
}
console.log(prototype.isPrototypeOf(teacher));              //true
```

propertyIsEnumerable方法用来检查对象的某个属性是否是可枚举的，也将返回布尔值，示例如下：

```
var prototype =   {
        subject:"JavaScript"
};
//检查对象的某个属性是否为可枚举的
console.log(prototype.propertyIsEnumerable("subject"));         //true
```

4-17 面向对象编程中的几个重要概念

1. 对象

对象是面向对象编程的核心，没有对象就没有办法面向对象。对象是可以实现某些功能的小单元，对象中会封装属性与行为。在ECMAScript语言中，行为也可以理解为一种属性。属性的实质就是变量或常量，只是其有特殊意义并与此对象相关，行为的实质就是方法，即函数，用来实现对象的某些功能。

2. 类

面向对象实际上也是自然生活的一种模拟，生活中存在各种各样的事物，一棵树是一个对象，一辆汽车是一个对象，每个人也是一个对象，等等。我们会将对象进行分门别类，例如生物下面分为动物和植物，动物下面又分为鱼类、鸟类、人类等，对于人类，我们又可以分为男人和女人，男人里面可以再细分，如老年、中年、青年和少年。将世间万物分类的基础在于同一类事物有统一的属性和类似的行为，例如鸟类都有翅膀，可以飞翔。这样的分类

可以使我们更轻松地了解自然，管理事物。在程序的世界里也是如此，前面我们一直在拿教师对象做示例，教师就可以定义为一个类，所有的教师都有姓名、年龄、所教科目这些属性，只是这些属性的值可能不同。

3. 封装

封装是将属性和行为捆绑在一起，创建对象的过程。封装也是一种抽象，例如实现生活中的汽车工厂，当原材料钢铁、橡胶、塑料等送入工厂后，经过流水线的生产，就会有一辆汽车从工厂中输出，这个过程就是一种封装，其核心思想是将复杂的过程封闭在系统的内部，对外界只提供入口和出口。

4. 继承

继承描述的是一种关系，其表示类的从属关系。在继承体系中，子类会继承父类的属性和行为，子类也可以重新定义自己的属性和行为，同样，子类也可以修改从父类继承来的属性和行为。这很像现实生活中的父子关系，孩子会继承父亲的一些外貌、性格等，但是孩子也有许多自己形成的个性。

5. 组合

简单理解，组合就是一个类，可以作为另一个类的属性。例如，一辆汽车会由很多子零件组成，轮胎是单独的类，引擎是单独的类，车体也是单独的类。这些类组合在一起构成更加复杂强大的汽车类。

6. 多态

不同对象对同一消息有不同的响应就是多态。举例而言，上课铃一响，所有教师都要开始教学动作，但是不同科目的教程所教的内容一定不同。上课铃可以理解为消息，其让教师执行教学动作，教学动作就是对象的响应，不同对象的响应不同。

4-18　用工厂方法模拟类

JavaScript当初在设计时只是为了执行一些简单的浏览器脚本，如今已经几乎无所不能，从前端到后端，使用JavaScript完成的项目越来越庞大。不幸的是，ECMAScript本身并不支持类，也就是说，ECMAScript中并没有类的概念。但是如果深入挖掘一下类的定义，你会发现在ECMAScript中模拟出类真的是十分灵活和简单的。

类说白了就是描述一类对象的通用属性，也可以理解为类是对象的模板。如果我们需要

一个教师对象，可以直接定义这个教师对象。但是如果我们需要很多教师对象，你可能已经想到了：我们可以定义一个工厂方法来生成教师对象。这样，在需要教师对象的时候，不需要再重新定义，调用这个工厂方法生成一个教师实例对象即可。这个工厂方法实际上就起到了类的作用，在ECMAScript中，这种函数被称为构造函数。

需要注意，在很多面向对象语言中，类是对象的基础，对象是由类构造出来的。类名一般要首字母大写，对象名首字母小写，这是一种编程规范。

ECMAScript中模拟类的方法有很多种，如果有兴趣，你也可以自己创造一种。本书将向你介绍4种模拟类的方法来抛砖引玉，希望可以帮你打开更宽广的思路。

工厂方法实际上也是函数，定义一个函数，在函数内部创建一个空对象，并对它进行一些类实例的完善，示例如下：

```
//模拟类
function Teacher(name,age,subject){
        var obj = {};
        obj.name = name;
        obj.age = age;
        obj.subject = subject;
        obj.teaching = function(){
                console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
        }
        return obj;
}
var jaki = Teacher("Jaki","25","JavaScript");
var lucy = Teacher("Lucy","24","Swift");
jaki.teaching();          //我是 Jaki,欢迎大家来听 JavaScript 教学课程。
lucy.teaching();          //我是 Lucy,欢迎大家来听 Swift 教学课程。
```

4-19　使用构造方法模拟类

上一示例中的工厂方法其实并非是真正的构造方法，前面我们说过，构造方法可以使用new关键字来调用，可以巧用其中this的指向来构造出对象，使用构造方法模拟创建一个教师类，示例如下：

```
//模拟类
function Teacher(name,age,subject){
        this.name = name;
        this.age = age;
```

```
        this.subject = subject;
        this.teaching = function(){
        console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
        }
    }
    var jaki = new Teacher("Jaki","25","JavaScript");
    var lucy = new Teacher("Lucy","24","Swift");
    jaki.teaching();        //我是 Jaki,欢迎大家来听 JavaScript 教学课程。
    lucy.teaching();        //我是 Lucy,欢迎大家来听 Swift 教学课程。
```

前面介绍过new关键字，当使用new关键字调用一个函数时，首先创建一个空对象，将这个对象与构造方法中的this进行绑定并建立原型链，之后执行构造方法进行对象的构造，最后将此对象返回。

4-20　使用 Object 函数对象的 create 方法模拟类

在前面的示例中，你已经学习了Object构造方法对象中的create函数，这个函数的作用是创建一个以某个对象为原型的对象。我们可以使用这个方法来模拟类，代码如下：

```
    var Teacher = {
        name:"Jaki",
        age:25,
        subject:"JavaScript",
        teaching:function(){
            console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
        }
    }
    var jaki = Object.create(Teacher);
    jaki.teaching();        //我是 Jaki,欢迎大家来听 JavaScript 教学课程。
```

使用这种方式来模拟类有一个弊端，在构造实例对象的时候没办法传入参数，如果要对对象的属性进行赋值，就必须在构造对象后再单独进行设置。

4-21　使用封装法模拟类

封装法模拟类的思路来源于其他面向对象语言的启发。示例如下：

```
var Teacher = {
    init:function(name,age,subject){
        var teacher = {};
        teacher.name = name;
        teacher.age = age;
        teacher.subject = subject;
        teacher.teaching = function(){
            console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
        }
        return teacher;
    }
};
var jaki = Teacher.init("Jaki",25,"JavaScript");
var lucy = Teacher.init("Lucy",24,"Swift");
jaki.teaching();        //我是 Jaki,欢迎大家来听 JavaScript 教学课程。
lucy.teaching();        //我是 Lucy,欢迎大家来听 Swift 教学课程。
```

使用封装法模拟类有一些先天的优势，其可以很方便地在类中定义私有属性和私有方法，也可以区别于实例方法和属性很容易地定义类方法和类属性。

在许多面向对象语言中都有实例属性、实例方法与类属性、类方法的定义，有些语言也会用静态属性和静态方法来描述类属性、类方法（如Swift），实例属性与实例方法是绑定在实例对象上的，类属性与类方法则是直接绑定在类上的。

4-22　使用对象冒充的方式实现继承

在ECMAScript中，实现继承机制的方式也多种多样。和模拟类的学习过程相似，本书会向你介绍3种实现继承机制的方法，分别为对象冒充法、原型法和混合法。希望你学习了这些方法后可以打开思维，更深入地理解ECMAScript语言的精髓。

对象冒充的方式利用了ECMAScript中this关键字的特性，你一定还记得，this关键字在函数内出现时，表示所调用此函数的对象。以教师类为例，首先所有的教师都是人类，人类都有年龄和姓名，因此，我们可以创建一个People类作为教师类的父类，示例代码如下：

```
//创建 People 类作为父类
function People(name,age){
    this.name = name;
    this.age = age;
}
```

```
function Teacher(name,age,subject){
    //这一步的作用是转换 this 的指向
     this.init = People;
     this.init(name,age);
     delete this.init;
    //添加教师类特有的属性
     this.subject = subject;
     this.teaching = function(){
          console.log("教师"+this.name+"正在教授"+this.subject+"课程。");
     };
}
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.teaching();        //教师 Jaki 正在教授 JavaScript 课程。
lucy.teaching();        //教师 Lucy 正在教授 Swift 课程。
```

上面的示例代码创建了一个最简单的继承体系，Teacher实例对象成功继承了People类中定义的姓名和年龄属性。其实可以将上面的代码进行简化，改变函数内this的指向，使用call函数和apply函数都可以做到，示例如下：

```
function People(name,age){
     this.name = name;
     this.age = age;
}
function Teacher(name,age,subject){
     People.call(this,name,age);
     this.subject = subject;
     this.teaching = function(){
          console.log("教师"+this.name+"正在教授"+this.subject+"课程。");
     };
}
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.teaching();        //教师 Jaki 正在教授 JavaScript 课程。
lucy.teaching();        //教师 Lucy 正在教授 Swift 课程。
```

要实现多继承也十分容易，示例如下：

```
function People(name,age){
     this.name = name;
     this.age = age;
```

```
    }
    function Work(time){
        this.time = time;
    }
    function Teacher(name,age,subject){
        People.call(this,name,age);
        Work.call(this,8);
        this.subject = subject;
        this.teaching = function(){
            console.log(" 教 师 "+this.name+" 正 在 教 授 "+this.subject+" 课 程 。 "+" 工 作 时 间 ：
"+this.time+"小时。");
        };
    }
    var jaki = new Teacher("Jaki",25,"JavaScript");
    var lucy = new Teacher("Lucy",24,"Swift");
    jaki.teaching();        //教师 Jaki 正在教授 JavaScript 课程。工作时间：8 小时。
    lucy.teaching();        //教师 Lucy 正在教授 Swift 课程。工作时间：8 小时。
```

上面的代码创建了一个人类和一个职业类，教师类既有人类的属性又有职业类的属性。

使用对象冒充的方式实现继承体系十分简单，但是严格来讲，其并非是真正意义上的继承，只是巧用了ECMAScript中this的特性。使用原型链的方式会实现更加类似于传统意义上的继承。还有一点需要注意，多继承是指一个子类可以有多个父类。C++是支持多继承的一种语言，Java、Objective-C、Swift等语言并不支持这种特性。

4-23　使用原型链的方式实现继承

原型链是ECMAScript中最难理解也是最令人费解的一部分。为了便于理解，我们可以再来研究一下new关键字到底做了什么。首先，在ECMAScript中，你要始终保持除原始值外万事万物都是对象这样一种思想，函数也是一种对象。当创建函数时，首先会创建这个函数对象本身，除此之外，还会创建一个对象作为此函数对象的prototype属性，例如：

```
    function People(){
        this.sayHi=function(){
            console.log("Hello,I am "+this.name+","+this.age+" years old。");
        }
    }
    console.log(People.prototype);        //People {}
```

你一定奇怪，这个People函数的prototype为什么会打印出People {}这样的信息。其实这只是一种格式化的输出，函数的prototype就是一个普通的Object对象，这个对象中自动生成了一个属性constructor，constructor默认指向当前的函数对象（即People）。当使用new关键字调用People方法进行实例对象的构造时，会以如下步骤进行：

步骤01 创建空对象{}，暂且命名为 obj。

步骤02 将构造函数中的 this 指向 obj，并将 obj 的__proto__属性指向构造函数的 prototype 属性，建立原型链。

步骤03 执行构造函数（prototype 中的 constructor 指向的函数，默认为函数本身）。

步骤04 将对象 obj 返回。

这里还有必要介绍一些__proto__属性，其实是Object实例对象中的一个私有属性，这个属性指向当前对象的原型对象。

上面的文字描述可能不是很容易理解，图4-1比较直观地解释了原型链的原理。

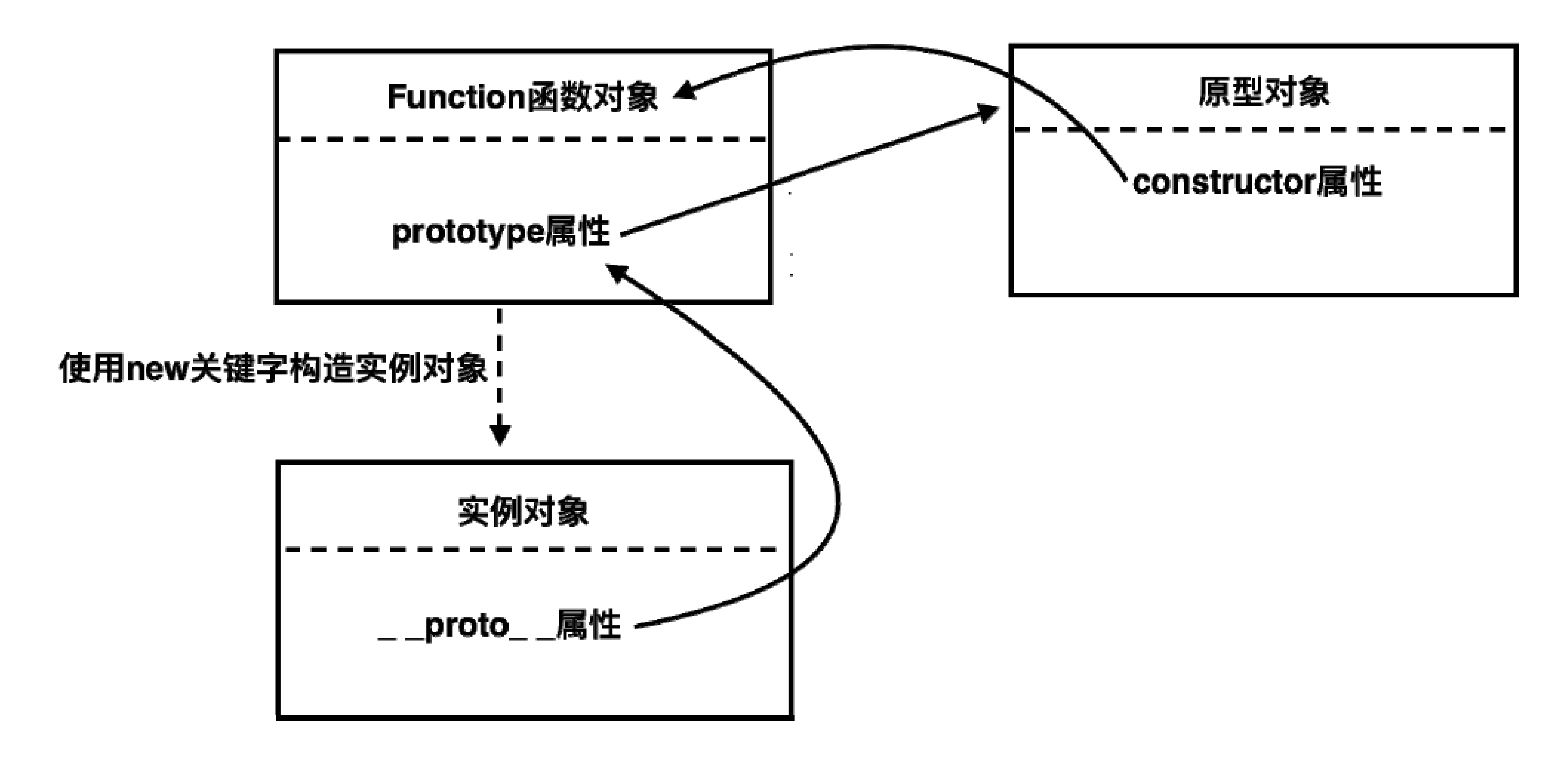


图 4-1　原型链示意图

使用原型链的方式实现继承，示例代码如下：

```
function People(){
    this.sayHi=function(){
        console.log("Hello,I am "+this.name+","+this.age+" years old。");
    }
}
function Teacher(name,age,subject){
    this.name = name;
```

```
        this.subject = subject;
        this.age = age;
    }
    //设置 prototype 属性
    Teacher.prototype = new People();
    var jaki = new Teacher("Jaki",25,"JavaScript");
    var lucy = new Teacher("Lucy",24,"Swift");
    jaki.sayHi();           //Hello,I am Jaki,25 years old。
    lucy.sayHi();           //Hello,I am Lucy,24 years old。
```

上面的代码在Teacher实例对象中并没有sayHi方法，在代码执行时，首先会在Teacher实例对象内部寻找这个方法，如果没有找到，会从实例对象的__proto__属性中寻找，这个属性其实就是Teacher函数对象的prototype指向的对象，即People实例对象。在这个People实例对象中找到了sayHi方法，将此方法返回，如果没有找到，再从People实例对象的__proto__属性里找，一层一层递归下去，直到找到或者最终对象的__proto__属性为undefined。访问属性时也是一样的原理，这便形成了一套完整的原型链。

使用原型链实现继承有一个很大的优势，其可以使用instanceof关键字来检查某个构造函数是否在对象的原型链上，这类似于检查某个对象是否为某个类或其子类的实例，示例如下：

```
    console.log(jaki instanceof People);          //true
```

当然，原型链的继承模式也有一些缺点，比如原型链是单链，无法实现多继承。通过原型链继承来的子类的实例在访问属性时可能会付出一定的性能做代价，因为其总是递归遍历从下往上查找属性的，如果访问一个undefined的属性，此性能代价会更大。

最后提一点，在ECMAScript中没有严格的私有属性的定义，这里说的私有属性其实是编码习惯上的，在开发中，我们常常把不想让外界访问的属性使用双下画线开头和结尾，将其作为一种编码规范上的私有。

4-24　使用混合模式实现继承

对象冒充的方式实现的继承归根到底是一种假继承，实质上所有实例对象都有一套完整的属性和方法，一般不同对象的属性值各有差异，每个对象都有一套独立的属性这点倒不是问题，问题在于所有同类的实例对象方法都是一致的，每个对象都有独立的一套就失去了继承的意义。原型链的方式实现继承时，父类的方法是放在原型对象中的，并不在对象本身里，因此所有同类的实例对象都是共享这些方法的，但是也有一个致命的缺陷，这种方法无法在

原型对象创建时指定参数，这也是为什么上面的示例中将教师姓名、年龄属性放在子类中而不是父类中。那么如何实现继承才是最优的模式呢？答案很简单，将属性使用对象冒充的方式来继承，方法使用原型链的方式来继承。示例如下：

```
function People(name,age){
        this.age = age;
        this.name = name;
}
//方法都放入原型链中
People.prototype = {
        constructor:People,
        sayHi:function(){
                console.log("Hello,I am "+this.name+","+this.age+" years old。");
        }
}
function Teacher(name,age,subject){
        //用对象冒充把属性继承过来
        People.call(this,name,age);
        this.subject = subject;
}
Teacher.prototype = new People();
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.sayHi();              //Hello,I am Jaki,25 years old。
lucy.sayHi();              //Hello,I am Lucy,24 years old。
```

这种混合模式实现的继承也是基于原型链的，因此可以使用instanceof关键字判断对象是否是某个类或其子类的实例。许多编译型语言对语法都有着严格的控制，相比之下，ECMAScript真的是一种非常灵活的语言，只要敢想，动手去尝试，你会发现其中乐趣无限。

4-25　编程练习

练习1：使用数组的排序方法实现数组元素的乱序。

解析：

```
var a = [1,2,3,4,5,6,7,8,9,0,12,23,45,90];
a.sort(function (a,b) {
       var sign = (Math.random()>0.5) ? 1 : -1;
```

```
        return (a-b)*sign;
});
```

练习2：编写函数，去除数组中的重复元素。

解析：

```
var a = [1,2,2,2,5,2,7,8,9,0,12,23,45,90];
function deRepeat(arr) {
    let newArr = [];
    arr.map(function (item) {
        if(newArr.indexOf(item) === -1){
            newArr.push(item);
        }
    });
    return newArr;
}
console.log(deRepeat(a));            //[ 1, 2, 5, 7, 8, 9, 0, 12, 23, 45, 90 ]
```

练习3：编写一个函数，输入字符串，判断字字符串中是否存在连续的字母（a~z或A~Z）连续出现。

解析：

```
function containsRepeatingLetter(str) {
    return /([a-zA-Z])\1/.test(str);
}
```

上面的正则表达式使用了新的语法，“（）”用来进行分组，“\1”表示对第1个分组进行引用，上面的正则描述重复出现两次相同的字母。

练习4：分析这行代码执行后的结果：["1", "2", "3"].map(parseInt);。

解析：

不运行代码要答对本题还是有相当的难度。数组的map函数用来进行数组的遍历并且执行回调，回调可以接收3个参数，分别为当前遍历到的元素值、元素索引和数组对象本身。parseInt是JavaScript中的一个全局函数，其作用是将字符串转换成数值，这个函数可以接收两个参数，第1个参数为要转换的字符串，第2个参数指定进制（2～36之间），如果参数不合法，将会返回NaN，所以上面的代码实际上是执行如下3条语句：

```
parseInt('1', 0);
parseInt('2', 1);
parseInt('3', 2);
```

最终的结果为：

```
[1, NaN, NaN]
```

练习5：编写函数，要求传入两个数组，进行合并后返回新的数组。

解析：

```
function func(arr1,arr2){
        return arr1.concat(arr2);
}
console.log(func([1,2,3],[4,5,6]));
```

练习6：编写函数，要求移除数组中的某个元素。

解析：

```
function func(arr,item){
          for(var i = 0; i < arr.length; i++){
              if(arr[i] === item){
                     arr.splice(i,1);
                     i--;
              }
       }
}
var arr =    [1,2,3,4,5];
func(arr,2);
console.log(arr);          //[1,3,4,5]
```

练习7：编写函数，找出某个元素在数组中的位置，并且返回此位置，如果没有找到，就返回-1。

```
function func(arr,item){
  for(var i = 0; i < arr.length; i++){
              if(arr[i] === item){
                     return i;
              }
       }
       return -1;
}
var arr = [1,2,3,4,5];
console.log(func(arr,3));               //2
```

练习8：编写函数，将使用“-”进行分割的字符串转换为驼峰风格，例如hello-world转换成helloWorld。

解析：

```
function func(str){
    var arr=str.split("-");
        for(var i=1;i<arr.length;i++){
            arr[i]=arr[i].charAt(0).toUpperCase()+arr[i].substring(1);
        }
    str=arr.join("");
    return str;
}
console.log(func("hello-world-js"));            //helloWorldJs
```

练习9：尝试编写一个函数，计算未来的某一天距离今天还剩多少天。

解析：

```
function func(date){
    var today = new Date();
    var l = date.getTime() - today.getTime();
    return Math.floor(l/1000/60/60/24);
}
console.log(func(new Date(2020,2,27)));
```

第 5 章

ECMAScript 的高级特性

通过本书前几章的学习，相信你已经对JavaScript的核心语法（ECMAScript）有了一定的了解和使用能力，解决常见的编程问题应该已经不在话下了。本章进一步学习ECMAScript的高级特性，这些功能与开发技巧可以让你的编程体验更加轻松畅快。

5-1 数组的解构赋值

解构赋值是ES6（ECMAScript 6）中一个十分强大的特性。掌握了这种技术，你将能十分轻松地从数组、对象等结构中提取所需要的数据。通俗地理解，解构赋值就是分解数据的结构来为变量赋值。这种语法特性广泛应用于数组的分解、对象的分解、字符串的分解、函数参数的分解等。

你一定还记得，数组实际上是一种特殊的对象，其属性名是递增的整数。如果我们要从数组中取值，通常会采用如下方式：

```
let students = ["Jaki","Lucy","Mery","July"];
//取数组中第 1 个元素
```

```
let stu1 = students[0];           //Jaki
//取数组中第 2 个元素
let stu2 = students["1"];             //Lucy
```

如果我们需要把数组中所有的值都提取到对应的变量中，上面的方法是十分麻烦的，需要手动地声明每个变量，并对其进行赋值。如果使用解构赋值技术，问题将变得十分简单，示例如下：

```
//进行数组的解构赋值
let [a,b,c,d] = students;
console.log(a+" "+b+" "+c+" "+d);              //Jaki Lucy Mery July
```

上面的代码将数组中的值按顺序提取到了a、b、c、d变量中。需要注意，解构赋值前面的let关键字和正常的变量声明意义一致，因此如果解构赋值中新声明的变量在前面有声明过，程序就会抛出异常（当然用var关键字不会有这个问题）。上面的示例代码students数组中有4个元素，进行解构赋值的表达式中也声明了4个变量，数组中的元素刚好可以和解构赋值表达式中的变量一一对应，这种解构赋值的场景叫作完全解构。与其相对，如果解构赋值表达式中声明的变量与数组中的元素个数并不一一对应，就会产生不完全解构，示例如下：

```
//不完全解构
//只提取数组中的前三个数据
let [e,f,g] = students;
console.log(e+" "+f+" "+g);          //Jaki Lucy Mery
//只提取数组中的第 4 个数据
let [,,,h] = students;
console.log(h);          //July
//提取数组中的第 1 个值，并将其余值放入另一个数组
let [i,...j] = students;
console.log(i+" "+j);           //Jaki [Lucy,Mery,July]
//溢出的变量将被赋值为 undefined
let [k,l,m,n,o] = students;
console.log(k+" "+l+" "+m+" "+n+" "+o);             //Jaki Lucy Mery July undefined
```

数组的解构赋值也支持进行嵌套，只要解构表达式的嵌套结构与数组的嵌套结构一致，就可以解构赋值成功，例如：

```
//解构赋值的嵌套
let array = [1,2,[5,6,7]];
let [p,q,[r,s,t]] = array;
console.log(""+p+q+r+s+t);          //12567
```

如果解构表达式声明的变量有溢出，就会解构失败，解构失败的变量将默认被赋值为undefined。在解构表达式中，你也可以为变量设置一个默认值，当解构失败或者解构出的值为undefined时，此变量会采用设置的默认值作为自己的值，示例如下：

```
//设置默认值
let [u=0,v=0,w=0] = [1,undefined];
console.log(u+" "+v+" "+w);        //1 0 0
```

需要注意，必须解构失败或者解构出的值严格为undefined时，变量才会采用默认值，其他解构出的如null、false、NaN等都不会触发变量默认值，例如：

```
[u=0,v=0,w=0] = [1,NaN,null];
console.log(u+" "+v+" "+w);        //1 NaN null
```

如上面的代码所示，其实在进行解构赋值时，并不一定要声明新的变量，也可以将数组中的数据解构赋值到已经存在的变量中。但是需要注意，如果是对象的解构赋值，就必须将解构赋值表达式放入小括号中，后面会具体介绍。

5-2　对象的解构赋值

和数组的解构赋值类似，对象也可以进行解构赋值。使用对象的解构赋值可以方便地提取对象的属性，例如：

```
//对象的解构赋值
let teacher = {
    name:"Jaki",
    age:25,
    teaching:function(){
        console.log("teaching...");
    }
};
let {name,age,teaching} = teacher;
console.log(name+" "+age); //jaki 25
teaching();        //teaching...
```

对象中的属性是没有先后顺序之分的，因此在对对象进行解构赋值时，赋值的变量名必须和对象中的属性名完全一致，否则会解构失败。如果要自定义解构赋值的变量名，可以采用如下映射方式：

```
let {name:myName,age:myAge} = teacher;
console.log(myName+" "+myAge); //jaki 25
```

其实对象的解构赋值的默认结构是这样的：

```
let {name:name,age:age} = teacher;
```

当要解构赋值的变量名与对象的属性名相同时，可以省略映射结构。数组也是对象，因此可以使用如下方式对数组进行解构：

```
let {0:x,1:y} = [1,2];
console.log(x+" "+y);        //1 2
```

对象的解构赋值也是支持嵌套的，这在处理复杂对象时十分高效，例如：

```
let teacher = {
    name:"Jaki",
    age:25,
    students:[
        {
            name:"Lucy",
            age:24
        },
        {
            name:"July",
            age:26
        }
    ],
    teaching:function(){
        console.log("teaching...");
    }
};
let {students:[{name:name1},{name:name2}]} = teacher;
console.log(name1+" "+name2);        //Lucy July
```

如果是将对象解构赋值到已有的变量中，解构赋值表达式就必须放在小括号内，否则会抛出异常，原因是JavaScript解释器会将大括号解析为代码块而不是表达式，例如下面的写法是正确的：

```
({name:name,age:age} = teacher);
```

在对象解构赋值时，也可以为变量添加默认值，如前面所说，只有当解构失败或者解构出的值为严格的undefined时，才会触发默认值。

5-3　字符串与函数参数的解构赋值

在进行字符串的解构赋值时，可以将字符串理解为一个字符数组，示例如下：

```
let [c1,c2,c3,c4,c5] = "Hello";
console.log(c1+c2+c3+c4+c5);            //Hello
```

这种解构赋值常用于提取字符串中某个位置的字符。在ES6中，函数的参数也可以进行解构赋值，在ES5标准中，如果你需要向函数中传递数组或对象，通常要这样做：

```
let people = {
    name:"Jaki",
    age:25
};
function print(people){
    console.log(people.name+":"+people.age);
};
print(people);          //Jaki:25
```

使用解构赋值的方式可以这样写：

```
let people = {
    name:"Jaki",
    age:25
};
function print({name,age}){
    console.log(name+":"+age);
};
print(people);          //Jaki:25
```

同样，你也可以为函数的参数在解构赋值时设置一个默认值。需要特别注意，为函数参数解构赋值设置默认值和设置函数参数的默认值是完全不同的，请看如下示例：

```
function print({name="name",age=0}={name:"Jaki",age:25}){
    console.log(name+":"+age);
};
print({});          //name:0
print();            //Jaki:25
```

上面的示例代码中，如果解构赋值失败或者解构为undefined，参数name会采用默认值“name”，参数age会采用默认值0。但是如果调用函数时不传入参数，就会采用参数对象的默认值，即{name:"jaki",age:25}对象，为函数的参数设置默认值也是ES6的特性之一。

5-4 用解构赋值交换变量的值

交换两个变量的值最少需要几步？你可能会不假思索地写出如下代码：

```
let v1=10;
let v2=11;
let v3 = v1;
v1 = v2;
v2 = v3;
console.log(v1);
console.log(v2);
```

上面的代码需要创建中间变量，并且需要至少3步才能完成两个变量值的交换，如果使用解构赋值技术，不仅可以省略中间变量，而且一步即可完成：

```
let v1=10;
let v2=11;
[v1,v2] = [v2,v1];
console.log(v1);
console.log(v2);
```

5-5 箭头函数的基本用法

箭头函数是ES6中新引入的一种创建函数的语法规则。这种语法可以很大程度地简化函数编写的格式，并且会对函数中的this指向进行绑定。

你一定还记得，前面我们介绍过好几种在ECMAScript中创建函数的方法，虽然几种创建函数的方法都有差别，但其在格式上大同小异，用法也基本上一致，例如：

```
function exp(a){
    return a*a;
}
```

```
let res = exp(5);
console.log(res);
```

如果使用箭头函数语法对上面的函数进行重写，结果如下：

```
let f = (a)=>{
    return a*a;
}
let res = f(5);
console.log(res);
```

箭头函数的基本语法格式为：(参数列表)=>{函数体}。如果箭头函数只有一个参数，并且函数体中只有一行代码，我们可以再进行简化，示例如下：

```
let f = a=>a*a;
let res = f(5);
console.log(res);
```

看到如此简洁的函数定义，是不是有眼前一亮的感觉呢！

需要注意，如果箭头函数的函数体只有一句代码，并且其返回的是一个对象，需要将对象包裹在小括号内，这是因为大括号默认会被解释为代码块，示例如下：

```
let func = ()=>({name:"Jaki"});
```

箭头函数在用法上，和我们前边介绍的普通函数并没有太大的差异，同样它也支持函数参数的解构赋值，示例如下：

```
let func = ({name,age})=>console.log(name,age);
func({name:"Jaki",age:25});        //Jaki 25
```

箭头函数这种简洁的结构十分适用于回调函数的编写，在实际发开发中，我们也会经常使用箭头函数。

5-6　箭头函数中 this 的固化

箭头函数有一个十分重要的特点，其中this是被固化的。在普通函数中，this默认指向调用函数的对象，当然也可以使用call、apply、bind这些函数进行this指向的更改。但是在箭头函数中，this是被固化的，也就是说，其中的this在定义函数时就已经被绑定，不能修改，也和调用者无关。比较下面两段代码。

普通函数：

```
let teacher = {
    name:"Jaki",
    age:25,
    print:function(){
        console.log(this.name,this.age);
    }
}
let student = {
    name:"Lucy",
    age:24,
    print:teacher.print
}
teacher.print();        //Jaki 25
student.print();        //Lucy 24
```

箭头函数：

```
//箭头函数 this 的固化
let teacher = {
    name:"Jaki",
    age:25,
    print:()=>{
        console.log(this.name,this.age);
    }
}
let student = {
    name:"Lucy",
    age:24,
    print:teacher.print
}
teacher.print();        //undefined undefined
student.print();        //undefined undefined
```

可以发现，在箭头函数中，this实际上并非指向teacher对象或者student对象，而是指向全局对象。即箭头函数内部的this就是定义时所在环境的this指向，并且会被固化，不能修改。再来看下面这个例子：

```
function foo(){
    this.name = "foo";
    this.inline = ()=>{
        console.log(this.name);
```

```
        };
        this.outline = function(){
            console.log(this.name);
        }
    }
    let obj = new foo();
    obj.inline();             //foo
    obj.inline.call({name:"Jaki"});          //foo
    obj.outline();            //foo
    obj.outline.call({name:"Jaki"});         //Jaki
```

从上面的代码可以看出，箭头函数在定义时，其中的this就固化称为foo函数当前环境中的this，无论调用方如何修改，这个this指向都不变。为了简单理解，我们可以将箭头函数中的this解析如下：

```
    function foo(){
        let _this = this;
        this.name = "foo";
        this.inline = ()=>{
            console.log(_this.name);
        };
        this.outline = function(){
            console.log(this.name);
        }
    }
```

由于箭头函数this固化的特性，因此不能将箭头函数作为构造函数来使用，即不可以使用new关键字来调用箭头函数。

需要注意，定义的全局对象中，箭头函数中的this之所以指向全局对象，是因为对象可以理解为是全局对象调用Object构造函数创建的，这个函数中的this当然指向其调用方全局对象，因此箭头函数中的this固化成了全局对象。

5-7　Set 集合结构

Set是一种集合结构，允许存储任意类型数据的唯一值。所谓唯一值，是指所存储的值不能重复。构造一个Set集合对象示例如下：

```
//创建 Set 集合
let set = new Set([1,2,3,4,4,2]);
console.log(set);          //Set { 1, 2, 3, 4 }
```

Set构造函数中可以传入一个数组对象，数组中的值会被作为Set集合的元素插入集合中。需要注意，数组中的元素如果有重复，就会自动被剔除，最终生成的Set集合中的元素都是唯一的。Set集合的这种特性也可以作为一种数组去重的好方法。Set实例对象的size属性可以获取集合中元素的个数，示例如下：

```
let set = new Set([1,2,3,4,4,2]);
console.log(set.size);          //4
```

关于Set实例对象中元素的操作，可以使用如下示例方法：

```
let set = new Set();
//向集合中插入元素
set.add("Jaki");
set.add("Lucy");
console.log(set);          //Set { 'Jaki','Lucy' }
//删除集合中的某个元素
set.delete("Jaki");
console.log(set);          //Set {'Lucy'}
//删除集合中所有元素
set.clear();
console.log(set);          //Set {}
set.add("Jaki");
set.add("Lucy");
//返回 Set 集合迭代器对象
console.log(set.entries());          //SetIterator { [ 'Jaki', 'Jaki' ], [ 'Lucy', 'Lucy' ] }
//让集合中的所有元素调用一次回调方法
/*
将打印
Jaki
Lucy
*/
set.forEach((element)=>{
    console.log(element);
},set);
```

```
//判断集合中是否包含某个元素
console.log(set.has("Jaki"));         //true
//下面这两个方法都是用来获取集合中所有元素的迭代器
console.log(set.keys());              //SetIterator { 'Jaki', 'Lucy' }
console.log(set.values());            //SetIterator { 'Jaki', 'Lucy' }
```

上面代码中的注释十分详尽，Set实例对象中的forEach方法与Array实例对象的forEach方法行为基本一致，都是将其中的元素遍历一遍，对每个元素执行传入的回调方法，forEach函数的第2个参数传入的对象会被绑定到回调函数中的this上。Set集合可以使用for-of结构进行迭代，示例如下：

```
/*
Jaki
Lucy
*/
for(item of set){
    console.log(item);
}
```

对象属性的遍历使用的是for-in结构，Set元素的遍历使用的是for-of结构，切记不要混淆。

ES6中还定义了一种特殊的Set集合：WeakSet。正如其名，它是一种弱引用集合。与Set集合相比，它有两个特点：

（1）WeakSet中只能存放引用数据，即对象数据，不能存放原始值。

（2）WeakSet中的对象元素都是弱引用的，因此其无法进行枚举。

WeakSet实例对象中提供的方法如下：

```
let obj1 = {name:"Jaki"};
let obj2 = {name:"Lucy"};
let wSet = new WeakSet([obj1]);
//弱引用集合中是否包含某个元素
console.log(wSet.has(obj1));          //true
//添加一个元素
wSet.add(obj2);
//删除一个元素
wSet.delete(obj1);
```

弱引用是指，除了集合之外，若没有其他变量或属性引用这个对象，则这个对象值会被垃圾回收机制回收掉，集合中的这个元素也将无效。

5-8　Map 字典结构

你一定有过查字典的经历。以汉语字典为例，当你需要查找某个字的释义时，需要先在字典的索引中找到这个字，然后根据索引提供的页标来找到这个字的释义。在字典结构中，我们常常把这个“索引字”称为键，把“释义”称为值。

许多编程语言中都有字典数据结构，在ES6标准前，ECMAScript中的对象可以充当字典来使用，但是对象的属性不能是任意类型的值，在ES6标准中引入了Map数据结构。Map就是简单的键值映射，其中键和值都可以是任意值。和Set数据结构相似的是，Map中的键也都是唯一的（不同键的值可以相同）。

Map实例对象中的size属性可以获取其中键值对的个数（去重后），代码如下：

```
//Map 字典
let map = new Map([["name","Jaki"],["age",25],[321,true],[321,true]]);
console.log(map.size);          //3
```

下面列出了一些常用的操作Map实例对象的方法：

```
let map = new Map();
//向 Map 实例对象中添加键值对
map.set("name","Jaki");
map.set("age",25);
map.set(123,true);
console.log(map);                 //Map { 'name' => 'Jaki', 'age' => 25, 123 => true }
//删除一对键值
map.delete(123);
console.log(map);                 //Map { 'name' => 'Jaki', 'age' => 25 }
//返回一个 Map 迭代对象
console.log(map.entries());       //MapIterator { [ 'name', 'Jaki' ], [ 'age', 25 ] }
//判断 Map 实例对象中是否包含某个键
console.log(map.has("name"));          //true
//获取 Map 中某个键的值，如果键不存在，就会返回 undefined
console.log(map.get("name"));          //Jaki
//获取 Map 中的所有键
console.log(map.keys());          //MapIterator { 'name', 'age' }
//获取 Map 中的所有值
console.log(map.values());        //MapIterator { 'Jaki', 25 }
/*
```

```
将打印
name Jaki
age 25
*/
map.forEach((value,key)=>{
    console.log(key,value);
},map);
//清空 Map 中所有键值
map.clear();
console.log(map);
```

Map实例对象调用forEach方法来对自身进行遍历，并对每个键值对执行回调方法，回调函数中，第1个参数为当前键值对的键，第2个参数为当前键值对的值。

Map数据结构也可以使用for-of结构来进行遍历，示例如下：

```
/*
name Jaki
age 25
*/
for(let [a,b] of map){
    console.log(a,b);
}
```

与WeakSet相对应，ES6中也定义了一个WeakMap。WeakMap是一种特殊的Map。其中所有的键只能是引用类型（即对象），值可以是任意类型。并且其对所有对象键的引用都是弱引用。因此，WeakMap也是不可以枚举的。WeakMap实例对象可用方法列举如下：

```
let wMap = new WeakMap();
let obj = {
    name:"Jaki"
}
//添加键值对
wMap.set(obj,true);
//判断某个键是否存在
console.log(wMap.has(obj));     //true
//获取某个键的值
console.log(wMap.get(obj));     //true
//删除一组键值对
wMap.delete(obj);
console.log(wMap.has(obj));     //false
```

5-9　使用 Proxy 代理对对象的属性读写进行拦截

ES6标准中定义了Proxy对象，从字面理解，它是一种“代理”对象。Proxy提供了一种途径，允许开发者对已经存在的对象行为进行拦截与修改。

首先创建一个对象，点语法可以轻松地访问对象的属性，例如：

```
let teacher = {
    name:"Jaki",
    age:25,
    teaching:function(){
        console.log("teaching");
    }
}
console.log(teacher.name);        //Jaki
```

下面我们使用Proxy对teacher对象属性的访问进行拦截：

```
let proxy_teacher = new Proxy(teacher,{
    set:(target,key,value,receiver)=>{
        console.log("添加属性:",key);
        target[key] = value;
    },
    get:(target,key,receiver)=>{
        console.log("获取属性:",key);
        return target[key];
    }
});
/*
将打印
获取属性: name
Jaki
添加属性: subject
获取属性: subject
JavaScript
*/
console.log(proxy_teacher.name);
proxy_teacher.subject = "JavaScript";
console.log(proxy_teacher.subject);
```

Proxy构造函数中需要传入两个参数，第1个参数为要代理的对象，第2个参数也是一个对象，其中需要定义要拦截或修改行为的方法，我们通常也会把它称为处理器对象。在对代理对象的某个属性赋值时，会触发处理器中的set方法，这个方法中前3个参数分别表示原对象、被赋值的属性名、所附的值。在读取代理对象的某个属性时会触发处理器中的get方法，这个方法中的前两个参数分别表示原对象和要访问的属性名。

需要注意，要使拦截方法起作用，必须使用Proxy代理对象，原对象的属性访问操作并不会触发代理对象的处理器方法。Proxy代理本身也是一种对象，对象可以通过原型链的方式来实现方法的继承，因此我们可以将Proxy代理作为对象的原型来实现未定义属性的拦截，示例如下：

```
var proxy_normal = new Proxy({}, {
   get: function(target, property) {
       console.log("warning:this property is undefined");
      return undefined;
   },
});
let obj = Object.create(proxy_normal);
obj.time;            //warning:this property is undefined
```

5-10　Proxy 代理处理器支持的拦截操作

上一示例我们演示了对属性读和写的拦截操作，分别在处理器中定义get和set方法即可。处理器支持的拦截操作不止如此，下面的示例代码列出了处理器对象中可以定义的拦截方法：

```
let teacher = {
        name:"Jaki",
        age:25,
        teaching:function(){
                console.log("teaching");
        }
}
let proxy_teacher = new Proxy(teacher,{
        //添加属性时会触发
        set:(target,key,value,receiver)=>{
                console.log("添加属性:",key);
                target[key] = value;
        },
```

```
    //获取属性时会触发
    get:(target,key,receiver)=>{
        console.log("获取属性:",key);
        return target[key];
    },
    //判断对象中是否包含某个属性时触发
    has:(target,key)=>{
        console.log("检查属性:",key);
        return key in target;
    },
    //删除对象属性时触发
    deleteProperty:(target,key)=>{
        console.log("删除属性:",key);
        delete target[key];
    },
    //拦截 getOwnPropertyNames()和 keys()方法
    ownKeys:(target)=>{
        console.log("获取所有自身属性");
        return Object.getOwnPropertyNames(target);
    },
    //拦截 defineProperty()和 definePropertys()方法
    defineProperty:(target,key,desc)=>{
        console.log("定义属性:",key);
        return Object.defineProperty(target,key,desc);
    },
    //拦截 preventExtensions()方法
    preventExtensions:(target)=>{
        console.log("抑制可扩展性");
        return Object.preventExtensions(target);
    },
    //拦截 getPrototypeOf()方法
    getPrototypeOf:(target)=>{
        console.log("获取对象原型");
        return Object.getPrototypeOf(target);
    },
    //拦截 isExtensible()方法
    isExtensible:(target)=>{
        console.log("获取对象可扩展性");
        return Object.isExtensible(target);
```

```
		},
		//拦截 setPrototypeOf()方法
		setPrototypeOf:(target,proto)=>{
			console.log("设置对象原型");
			return Object.setPrototypeOf(target,proto);
		},
		//拦截 call()和 apply 方法，用于函数对象
		apply:(target,object,arguments)=>{
			console.log("拦截 call、apply 方法");
			target.apply(object,arguments);
		},
		//拦截构造函数方法
		construct:(target,arguments)=>{
			return {};
		}
	});
	console.log("name" in proxy_teacher);		//检查属性：name true
	delete proxy_teacher.name //删除属性: name
	console.log(Object.getOwnPropertyNames(proxy_teacher)); //获取所有自身属性 [ 'name', 'age',
'teaching' ]
	Object.defineProperty(proxy_teacher,"name",{
		value:"Jaki",
		writable:true,
		configruable:true
	});		//定义属性: name
	// Object.preventExtensions(proxy_teacher);		//抑制可扩展性
	Object.getPrototypeOf(proxy_teacher);		//获取对象原型
	Object.isExtensible(proxy_teacher);		//获取对象可扩展性
	Object.setPrototypeOf(proxy_teacher,{sex:"男"});	//设置对象原型
```

需要注意，in是前面没有专门提到的一个运算符，使用它可以获取对象中是否包含某个属性。

关于使用Proxy来代理目标对象，有一点需要额外注意，Proxy对象方法中的this和原对象方法中的this并不一致，它们指向的是两个不同的对象，例如：

```
	var student = {
		name:"Lucy",
		print:function(){
			console.log(this===student);
		}
```

```
}
let proxy_student = new Proxy(student,{});
student.print();                //true
proxy_student.print();          //false
```

5-11　使用 Promise 承诺对象

Promise为ECMAScript异步编程提供了一种实现思路。比如某个延时任务，我们可以把它交给Promise对象，这样既不会阻塞程序的正常执行，当延时任务执行完成后，我们又可以第一时间做逻辑处理。代码示例如下：

```
//打印结果
/*
go...
任务直接完成
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务执行完成");
    },3000);
});
console.log("go...");
```

使用Promise构造函数来进行Promise实例对象的创建，构造函数中需要传入一个函数作为参数，这个参数函数比较特殊，它有两个参数，分别用来触发Promise实例对象的任务执行完成消息和Promise实例对象的任务执行失败消息，后面会具体介绍，参数函数的函数体就是Promise实例对象需要执行的任务代码。从上面的打印信息可以看出，Promise实例对象一旦被创建，其中的任务会自动开始执行，上面的代码做了延时3秒的打印操作，并没有阻塞主程序代码的执行。

其实Promise实例对象有3种状态，分别是pending（执行中）、fulfilled（执行完成）、rejected（执行失败）。这3种状态都是内部触发的，外部无法对其进行操作。上面的代码并没有对Promise任务执行完成情况进行监听，在实际开发中，一般不仅需要得到Promise任务执行的情况，甚至还需要获取任务执行完成后的一些数据。Promise实例对象的then方法用来监听执行情况并接收传递的数据。请看如下代码：

```
//打印结果
/*
go...
任务直接完成
Success
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务执行完成");
        resolve("Success");
    },3000);
});
promise.then((success)=>{
    console.log(success);
},(error)=>{
    console.log(error);
});
```

then方法中可以传入两个参数，第1个参数为当Promise任务执行完成时回调的方法，第2个参数为当Promise任务执行失败时回调的方法，第2个参数为非必需的。仅仅对Promise的执行情况进行监听并没有意义，我们还需要告诉Promise对象什么情况算任务执行完成，什么情况又算任务执行失败。在创建Promise对象时，我们提到过参数函数中的两个参数：resolve和reject。这两个参数实际上也是函数，我们只需要在函数体合适的地方对它们进行调用即可，例如上面的代码调用了resolve，这时Promise示例对象的状态被置为已完成，之后会执行then方法监听的已完成情况下的回调函数。

除了then方法外，Promise实例对象中要定义一个catch方法，catch方法和then方法的区别只在于catch方法只有一个参数，其是Promise执行失败后回调的函数，例如：

```
//打印结果
/*
go...
任务直接完成
error
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务直接完成");
        reject("error");
    },3000);
```

```
});
promise.catch((error)=>{
    console.log(error);
});
```

5-12　建立 Promise 任务链

在开发中还经常会遇到这样的场景，任务B的执行必须依赖于任务A，即只有当任务A成功执行后，才可以执行任务B。使用Promise可以十分容易地实现这种逻辑，其实Promise实例对象的then方法中可以返回一个新的Promise来构成任务链，示例代码如下：

```
//打印结果
/*
go...
任务直接完成
success
任务 2 执行完成
success2
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务执行完成");
        resolve("success");
    },3000);
});
promise.then((success)=>{
    console.log(success);
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log("任务 2 执行完成");
            resolve("success2");
        },2000)
    });
},(error)=>{
    console.log(error);
}).then(success=>{
```

```
        console.log(success);
});
console.log("go...");
```

使用这种任务链的编程模式可以很轻松地处理多任务并且有复杂依赖关系的场景。

5-13　进行 Promise 对象组合

Promise构造函数对象还提供了两个非常有用的方法，它们用来对Promise任务进行组合，all方法用来组合一组Promise实例对象，当组中所有的Promise任务都成功执行完成后，Promise任务组才算成功执行，如果其中有一个任务执行失败，任务组就会执行失败，例如：

```
let promise1 = new Promise((resolve,reject)=>{
        console.log("任务 1 执行成功");
        resolve();
});
let promise2 = new Promise((resolve,reject)=>{
        console.log("任务 2 执行成功");
        resolve();
});
let promise3 = new Promise((resolve,reject)=>{
        console.log("任务 3 执行成功");
        resolve();
});
let promiseGroup = Promise.all([promise1,promise2,promise3]);
/*
任务 1 执行成功
任务 2 执行成功
任务 3 执行成功
任务组执行成功
*/
promiseGroup.then(success=>{
        console.log("任务组执行成功");
},error=>{
        console.log("任务组执行失败");
});
```

如果我们将任务组中的一个单独任务修改为执行失败，结果就会完全不同，代码如下：

```
let promise1 = new Promise((resolve,reject)=>{
    console.log("任务 1 执行成功");
    resolve();
});
let promise2 = new Promise((resolve,reject)=>{
    console.log("任务 2 执行失败");
    reject();
});
let promise3 = new Promise((resolve,reject)=>{
    console.log("任务 3 执行成功");
    resolve();
});
let promiseGroup = Promise.all([promise1,promise2,promise3]);
/*
任务 1 执行成功
任务 2 执行失败
任务 3 执行成功
任务组执行失败
*/
promiseGroup.then(success=>{
    console.log("任务组执行成功");
},error=>{
    console.log("任务组执行失败");
});
```

与all方法对应的还有一个race方法，这个方法也是组合一组Promise实例对象，不同的是race方法组合的任务组中只要有一个任务执行完成，任务组就算执行完成，任务组的成功或失败只与第一个完成的任务有关，第一个完成任务的Promise对象状态如果为成功，任务组的状态就为成功，此对象的状态为失败，任务组的状态就为失败，例如：

```
let promise1 = new Promise((resolve,reject)=>{
    //加延时
    setTimeout(()=>{
        console.log("任务 1 执行成功");
        resolve();
    },1000);


});
let promise2 = new Promise((resolve,reject)=>{
```

```
        //加延时
        setTimeout(()=>{
                console.log("任务 2 执行成功");
                resolve();
        },1000);
});
let promise3 = new Promise((resolve,reject)=>{
        console.log("任务 3 执行失败");
        reject();
});
let promiseGroup = Promise.race([promise1,promise2,promise3]);
/*
任务 3 执行失败
任务组执行失败
任务 1 执行成功
任务 2 执行成功
*/
promiseGroup.then(success=>{
        console.log("任务组执行成功");
},error=>{
        console.log("任务组执行失败");
});
```

Promise构造函数对象的all方法返回的任务组的成功和失败取决于其中每一个任务的执行。Promise构造函数对象的race方法返回的任务组的成功和失败只与第一个执行完的任务有关，这样对比可以便于记忆。

5-14　Generator 函数应用

使用Promise可以十分轻松地编写链式任务，在ECMAScript中还提供了一种强大的任务执行控制方案：Generator生成器对象。

从字面上理解，Generator是一种生成器对象，这种理解并不确切，但也没错。一种更加面向应用的理解为Generator是一个状态机，其内部封装了多种状态，通过yield语句进行隔离。使用Generator函数分步处理任务将更加容易。

下面是一个简单的Generator函数例子：

```
function* generatorFunc(){
    console.log("任务一");
    yield;
    console.log("任务二");
    yield;
    console.log("任务三");
    return;
}
let g = generatorFunc();
let t1 = g.next();        //任务一
let t2 = g.next();        //任务二
let t3 = g.next();        //任务三
let t4 = g.next();
console.log(t1);          //{ value: undefined, done: false }
console.log(t2);          //{ value: undefined, done: false }
console.log(t3);          //{ value: undefined, done: true }
console.log(t4);          //{ value: undefined, done: true }
```

我们先来看Generator函数的语法，其和普通函数十分相似，不同的是，在function关键字后追加一个星号，表示它是一个Generator函数。Generator函数内部可以编写一些逻辑代码，和普通函数不同，除了可以使用return关键字返回外，还可以使用yield关键字来中断（普通函数中不可以使用yield关键字）。Generator函数的调用和普通函数一样，但是有一点需要特别注意，调用Generator函数并非执行函数体的内容，而是返回一个迭代器对象，通过这个迭代器对象的next指针来分步执行Generator函数体中的任务。

以上面的代码为例，g为调用Generator函数返回的迭代器对象，当迭代器对象第一次调用next方法时，会执行Generator函数体的代码，直到遇到yield语句处中断。第2次再调用next方法时，Generator函数体中的任务会从上次中断的地方开始继续执行，直到再次遇到yield中断或者return返回。当然，调用next函数时，本身也会有一个返回值，这个返回值是一个对象，其中的done属性如果为false，代表Generator函数体的任务还没有执行完成，可以继续调用next往后执行，如果done属性为true，则表示Generator函数体的任务已经完全执行完成，之后再次调用next方法将不会有效果。从上面代码的打印结果也可以看到，next函数返回的对象中还有一个value属性，这个属性用于接收Generator函数体任务执行中使用yield或者return返回的数据，例如：

```
function* generatorFunc(){
    console.log("任务一");
    yield 1;
    console.log("任务二");
```

```
        yield 2;
        console.log("任务三");
        return 3;
    }
    let g = generatorFunc();
    let t1 = g.next();        //任务一
    let t2 = g.next();        //任务二
    let t3 = g.next();        //任务三
    let t4 = g.next();
    console.log(t1);          //{ value: 1, done: false }
    console.log(t2);          //{ value: 2, done: false }
    console.log(t3);          //{ value: 3, done: true }
    console.log(t4);          //{ value: undefined, done: true }
```

再来看一种特殊的情况，如果我们要在一个Generator函数任务中执行另一个Generator函数任务，该如何来做呢？请看如下代码：

```
    function* generatorSubFunc(){
        console.log("子任务一");
        yield;
        console.log("子任务二");
        return;
    }
    function* generatorFunc(){
        console.log("任务一");
        yield 1;
        console.log("任务二");
        let sub = generatorSubFunc();
        sub.next();
        sub.next();
        yield 2;
        console.log("任务三");
        return 3;
    }
    let g = generatorFunc();
    let t1 = g.next();        //任务一
    let t2 = g.next();        //任务二  子任务一  子任务二
    let t3 = g.next();        //任务三
```

上面的示例代码提供了一种解决方案，但是代码结构并不优美，ES6中提供了yield*语

句来直接在Generator函数体内部执行另一个Generator函数体任务，修改上面的代码如下：

```
function* generatorSubFunc(){
    console.log("子任务一");
    yield;
    console.log("子任务二");
    return;
}
function* generatorFunc(){
    console.log("任务一");
    yield 1;
    console.log("任务二");
    yield* generatorSubFunc();
    console.log("任务三");
    return 3;
}
let g = generatorFunc();
let t1 = g.next();        //任务一
let t2 = g.next();        //任务二、子任务一、子任务二
let t3 = g.next();        //任务三
```

5-15 Generator 任务参数的传递

你知道通过yield或return语句可以得到Generator函数每一步任务的返回值，但是函数有三要素：定义域、值域、表达式。对应编程，其实就是参数、返回值、函数体。比如我们要实现这样功能的Generator函数：第一个任务先传入两个参数，计算其和，然后输出结果；第二个任务再计算上一步结果的平方，再次输出，最后一个任务再传入一个参数，计算上一步结果与传入参数的差，再输出。其实我们在调用next方法的时候，可以传递参数，配合yield语句来接收传递的参数，示例如下：

```
function* cal(){
    console.log("可以开始计算");
    let a = yield;
    console.log("接收参数 a",a);
    let b = yield;
    console.log("接收参数 b",b);
    let res = a+b;
```

```
        yield res;
        res = res*res;
        let d = yield res;
        console.log("接收参数 d",d);
        res = res - d;
        return res;
}
let c = cal();
c.next();
c.next(5);
console.log(c.next(3));        //{ value: 8, done: false }
console.log(c.next());         //{ value: 64, done: false }
console.log(c.next(10));            //{ value: 54, done: true }
```

如上面的代码所示，当调用next函数的时候，我们可以传入一个参数，此参数会作为上一个任务yield表达式的值，切记要注意，传入的参数会作为上一次任务的yield表达式的值，而不是本次任务yield表达式的值。也就是说，第一次调用next方法时，实际上是不能传入参数的。

当Generator函数作为对象的属性时，往往可以简写，如下面的代码所示：

```
var obj = {
        gFunc1:function* (){
                //...
        },
        * gFunc2(){
                //...
        }
}
```

obj对象中的gFunc1与gFunc2都是Generator函数。

5-16　使用 class 定义类

ECMAScript是一种面向对象语言，但是它并不是基于类的。通过前面的学习，你已经掌握了在ECMAScript中模拟类的方式，尽管我们可以通过对象冒充或者原型链的方式来实现类的功能，但这种定义类的方式与传统面向对象语言有很大不同，ES6中新引入了class关键字，使用它定义类将更加容易、更加规范。

先来回顾一下使用构造函数定义类的方式：

```
function Teacher(name,age){
    this.name = name;
    this.age = age;
    this.teaching = ()=>{
        console.log(this.name,this.age);
    }
}
let teacher = new Teacher("Jaki",25);
teacher.teaching();            //Jaki 25
```

ES6的class关键字定义了一种类模板，使用class关键字来改写上面的Teacher类如下：

```
class Teacher{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
    teaching(){
        console.log(this.name,this.age);
    }
}
let teacher = new Teacher("Jaki",25);
teacher.teaching();            //Jaki 25
```

下面解释一下class关键字的用法，首先class关键字用于定义一个类，在代码块内需要提供一个constructor方法作为类的构造方法，也就是说，当使用new关键字加类名来创建实例化对象时，系统会调用constructor方法。除了constructor方法外，还可以在类中添加其他自定义方法，这些方法默认都会放入实例对象的原型对象中，换句话说，这些方法是所有实例对象所共享的。

需要注意，你不一定非要显式地实现constructor方法，如果你不实现，系统就会自动提供一个空的constructor方法作为构造函数。

5-17　使用 class 实现类的继承

使用class关键字定义类的另一个方便之处在于，继承对其来说变得十分容易。我们不需要再手动对原型链进行操作，例如：

```
class People{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
}
class Teacher extends People{
    constructor(name,age,subject){
        super(name,age);
        this.subject = subject;
    }
    teaching(){
        console.log(this.name,this.age,this.subject);
    }
}
let teacher = new Teacher("Jaki",25,"JavaScript");
teacher.teaching();          //Jaki 25 JavaScript
```

class的继承采用extends关键字，需要注意，如果你使用了继承，那么在constructor构造方法中必须先调用父类的构造方法，即使用super关键字来调用。super关键字既可以当函数来使用，又可以当父类对象、原型对象来使用。若在子类的构造方法中使用super()，则表示调用父类的构造方法，若使用super.method的方式来调用父类的方法，则表示的是父类对象的原型对象，例如：

```
class People{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
    sayHi(){
        console.log("sayHi");
    }
}
class Teacher extends People{
    constructor(name,age,subject){
        super(name,age);
        this.subject = subject;
        super.sayHi();
    }
```

```
        teaching(){
            console.log(this.name,this.age,this.subject);
        }
    }
    let teacher = new Teacher("Jaki",25,"JavaScript");        //sayHi
    teacher.teaching();             //Jaki 25 JavaScript
```

使用class定义类可以实现的功能在我们前面介绍的模拟类的示例中基本也可以实现。其实，在ES6中，class这种语法只是引入了一种更加直观可读、更加规范的定义类的语法层面的实现，其实质依然是原型链。

5-18　认识 JSON 数据格式

JSON数据对于任何互联网编程人员来说都是十分亲切的，作为一种轻量级的数据交换格式，JSON在前后端通信中有着十分重要的价值。JSON全称JavaScript Object Notation，即JavaScript对象节点。它是基于ECMAScript定制的一种数据格式规范，但是和我们前面所了解的ECMAScript对象的定义规范并不完全一致。

在编写JSON数据字符串时，你需要遵守如下规则：

（1）用花括号来表示独享。
（2）用中括号来表示数组。
（3）对象中的属性用逗号分隔，键与值使用冒号分隔，并且键需要用双引号包裹。
（4）数组中的数据使用逗号分隔。

需要注意，JSON的本质是有着一定书写规则的字符串，其和ECMAScript中的对象有着本质的区别。在JSON规则中，可以描述的类型只有6种，列举如下：

（1）数值（整数或浮点数），可以直接编写。
（2）字符串，需要在双引号中。
（3）布尔值，true或者false。
（4）数组，用中括号包裹。
（5）对象，用大括号包裹。
（6）null，表示空值。

5-19　使用 JSON 对象

首先你需要清楚，本示例中所说的JSON对象是ECMAScript中一个内置对象，使用它可以很容易地实现ECMAScript对象与JSON字符串之间的转换。

将ECMAScript对象转换成JSON对象的示例如下：

```
var teacher = {
    name:'Jaki',
    age:25,
    teaching:function(){
        console.log("teaching");
    }
}
var json = JSON.stringify(teacher);
console.log(typeof json,json);          //string {"name":"Jaki","age":25}
```

需要注意，JSON语法中并没有函数这种数据类型，ECMAScript对象中的方法会被默认剔除。我们再来深入研究一下JSON对象的stringify方法，这个方法的主要作用是将对象转换成JSON字符串，上面的示例代码是stringify方法的最简单使用，其实这个方法中还可以有另外两个参数，完整格式如下：

```
stringify(obj,rep,spac)
```

其中，obj参数是我们要进行转换的对象，rep参数是可选的，若我们将其以数组的形式进行设置，则在进行字符串构建时，只有数组中有的属性会被解析，例如：

```
var teacher = {
    name:'Jaki',
    age:25,
    teaching:function(){
        console.log("teaching");
    }
}
var json = JSON.stringify(teacher,["name"]);
console.log(typeof json,json);          //string {"name":"Jaki"}
```

若传入的rep参数为函数，则此函数用来进行JSON值解析的处理。例如，我们可以让解析出的所有字符串的值转换为大写，例如：

```
var teacher = {
        name:'Jaki',
        age:25,
        teaching:function(){
                console.log("teaching");
        }
}
var json = JSON.stringify(teacher,function(key,value){
        if (typeof value === 'string') {
                return value.toUpperCase();
        }
        return value;

});
console.log(typeof json,json);              //string {"name":"JAKI","age":25}
```

stringify方法的第3个参数用来进行格式化输出，如果不传，生成的JSON字符串属性前就无空格；如果传入1～10之间的一个数值，就表示使用多少个空格进行格式化；如果传入的参数是一个字符串，就使用此字符串进行分割，例如：

```
var teacher = {
        name:'Jaki',
        age:25,
        teaching:function(){
                console.log("teaching");
        }
}
/*
{
• • • "name": "JAKI",
• • • "age": 25
}
*/
var json = JSON.stringify(teacher,function(key,value){
        if (typeof value === 'string') {
                return value.toUpperCase();
        }
        return value;

},"…");
```

相对于将ECMAScript对象转换为字符串，JSON对象中的parse方法则可以将一个JSON字符串转换成ECMAScript对象，例如：

```
var obj = JSON.parse("{\"name\":\"Jaki\",\"age\":25}");
console.log( typeof obj,obj);        //object { name: 'Jaki', age: 25 }
```

同样，parse方法中也可以传入两个参数，第2个参数为回调函数，用来处理JSON字符串中解析出来的属性。例如，我们想把上面示例JSON字符串中的name属性的值修改为Lucy，可以这样写：

```
var obj = JSON.parse("{\"name\":\"Jaki\",\"age\":25}",function(key,value){
        console.log(key,value);
        if (key === 'name') {
                return "Lucy";
        }
        return value;
});
console.log( typeof obj,obj);        //object { name: 'Lucy', age: 25 }
```

5-20　认识 Symbol

你一定还记得，前面我们说在ECMAScript中的原始值类型只有5种。其实这并不完全正确，将其修改为“传统上来说，ECMAScript中的原始值类型只有5种”会更加确切一些。在ECMAScript 6中暴露了Symbol类型给开发者，其也是一种原始值类型，并且存在的唯一用途就是作为对象的属性名符号。

目前，你一定是一头雾水，属性名符号是个怎样的概念呢？其实也很简单，例如，我们在构建对象时通常会采用下面的方式：

```
var obj = {
        name:"Jaki"
}
obj.age = 25;
obj["subject"] = "JavaScript";
console.log(obj);        //{ name: 'Jaki', age: 25, subject: 'JavaScript' }
```

在上面的代码中，name、age、subject都可以理解为一种属性名符号，只是这些是传统的属性名符号，Symbol是一种特殊的属性名符号。

Symbol并不是构造函数，直接使用Symbol函数来创建Symbol类型的值，例如：

```
var symName = Symbol("name");
var symAge = Symbol("age");
var teacher = {
    [symName]:"Jaki"
}
teacher[symAge] = 25;
console.log(teacher[symName],teacher[symAge]);        //Jaki 25
```

上面的代码中有几点需要注意，首先Symbol属性不能使用点语法，其次Symbol属性有一定的不可见性。也就是说，我们使用for-in对对象的属性进行遍历或者使用keys、getOwnPropertyNames方法来获取对象的属性名都无法获取Symbol属性，Object函数对象上封装的getOwnPropertySymbols函数专门用来获取对象上的Symbol属性，例如：

```
console.log(Object.getOwnPropertySymbols(teacher));        //[ Symbol(name), Symbol(age) ]
```

还有一点需要注意，上面的代码在创建Symbol值的时候有传入一个字符串参数，其实这个参数并没有真正的用途，只用来标识符号，帮助我们进行打印调试。

5-21　注册全局的 Symbol 符号

在调用Symbol函数创建Symbol值的时候，每次都会创建一个唯一的符号值，例如：

```
var s1 = Symbol("symbol");
var s2 = Symbol("symbol");
console.log(s1 === s2);            //false
```

从上面的代码可以看出，传入的描述参数并不会起到复用符号的作用。Symbol对象中提供了方法用来注册全局的Symbol符号和从全局注册表中获取Symbol符号，示例代码如下：

```
//注册
var s3 = Symbol.for("key");
var s4 = Symbol.for("key");
console.log(s3,s4,s3===s4);        //Symbol(key) Symbol(key) true
console.log(Symbol.keyFor(s3));        //key
```

首先，Symbol对象中的for方法并不总是会返回一个新的符号值，首先会从注册符号表中找是否已经存在这个符号值，如果存在，就直接返回，如果不存在，才会新建并放入符号表。从打印信息可以看出，全局符号是对应key值唯一的。Symbol对象中的keyFor方法则是用来获取某个全局符号对应在符号表中的键值。

5-22　迭代器 Symbol

我们知道，有些对象可以使用for-of进行迭代，像数组对象，有些对象却不行。原因是可以使用for-of进行迭代的对象都实现了迭代器方法。我们如何让一个自定义的对象支持for-of迭代呢？使用iterator符号来定义方法即可，例如：

```
var myArray = {
    name:'Jaki',
    age:25,
    [Symbol.iterator]:function* (){
        yield 'Jaki';
        yield 25;
    }
}
//将打印 Jaki 25
for(let v of myArray){
    console.log(v);
}
//打印  Jaki 25
console.log(...myArray);            //同  console.log('Jaki',25);
```

需要注意，迭代器符号需要设置为一个迭代器对象，而Generator函数正是会生成一个迭代器对象。还有一点需要注意，“...”也是一种运算符，其用来进行可迭代对象的展开。

5-23　正则表达式符号

在学习字符串类型的过程中，你是否也有过疑问，正则表达式是如何进行匹配操作的？其实使用正则表达式符号，任何自定义的对象都可以起到正则表达式的作用，例如字符串实例对象中的match、replace、search、split方法。默认情况下，我们只能将字符串或者正则表达式对象当成参数传入，如果我们自定义的对象实现了相关符号属性，也可以作为参数传入这些方法，例如：

```
var myArray = {
    name:'Jaki',
    age:25,
```

```
        [Symbol.iterator]:function* (){
            yield 'Jaki';
            yield 25;
        },
        //match
        [Symbol.match]:function(string){
            return string+"match";
        },
        [Symbol.replace]:function(string){
            return string+"replace";
        },
        [Symbol.search]:function(string){
            return string+"search";
        },
        [Symbol.split]:function(string){
            return string+"split";
        }
    }
    console.log("subject".match(myArray));          //subjectmatch
    console.log("subject".replace(myArray));        //subjectreplace
    console.log("subject".search(myArray));         //subjectsearch
    console.log("subject".split(myArray));          //subjectsplit
```

最后，还有一点需要注意，Symbol属性在进行JSON转换的时候会被自动忽略。

5-24　使用 export 进行模块的导出

在ECMAScript中，一直都缺少一种模块引入的语法。这使得大项目的拆分变得十分困难。ES6中引入了模块化的设计思想，通过export关键字可以将一个文件内的对象导出，在另一个文件中，使用import可以实现其他文件中对象的导入。由于目前大多数浏览器的JavaScript解释环境并不能支持export和import模块编程，因此这里只做语法的演示。

export用于规定当前文件对外提供的接口，import则是引入这些功能接口。

例如，创建一个命名为module.js的文件，如果我们要将其作为配置文件向外界提供一些用户配置信息的接口，可以这么做：

```
export var userName = "Jaki";
export var password = "123456";
```

上面的示例代码在变量的声明前添加了export关键字，其向外部输出了userName和password两个变量。当然，你也可以选择在一次性导出多个变量，例如：

```
var scrool = "No.1";
var theClass = "No.2";
export {scrool,theClass};
```

除了可以将变量作为导出的对象外，也可以导出函数或者类，例如：

```
export function(){
    console.log("Hello World");
}
export class Teacher{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
}
```

默认情况下，导出的对象在外界的名称就是对象原始的名称，你也可以对导出对象的名称使用as进行自定义，例如：

```
var jaki = "Jaki";
var lucy = "Lucy";
export {jaki as people1,lucy as people2};
```

对于上面示例代码，外界在使用jaki和lucy变量时，需要使用people1与people2。

5-25　使用 import 进行模块的导入

import关键字和export关键字对应，其用于引入export关键字所导出的对象。我们再创建一个名为main.js的文件，比如要导入module.js中的userName和password变量，可以这么做：

```
import {userName,password} from "./module.js";
console.log(userName,password);
```

import后面的大括号中需要指定要引入模块中导出的对象，from关键字后面需要指定模块文件的路径。同样，在引入文件时，你也可以为引入的对象起一个新的名字，例如：

```
import {userName as name,password as word} from "./module.js";
console.log(name,word);
```

improt还提供了一种语法，其可以将外部模块中所有导出的对象一起引入进来，并且绑定到指定的对象上，例如：

```
import * as module from "module.js";
console.log(module.userName,module.password);
```

很多时候，一个外部的模块往往只需要向外提供一个唯一的接口。例如，通常将一个独立的类写成一个单独的文件，那么只暴露这个类本身作为接口就可以了。因此，ES6的导出与导入还提供了default默认项设置。例如，在module.js文件中编写如下代码：

```
export default class People(){
        constructor(){
        }
}
```

上面代码的意思是将People类作为module.js文件的默认导出对象。在main.js中，可以使用如下代码进行导入：

```
import myModule from "module.js";
```

上面的示例代码有两点需要注意，首先myModule是自定义的名称，代表module.js文件中默认导出的对象，即People类。还有一点需要特别注意，这种当时引入的对象是不需要写在大括号内的。默认对象和非默认对象也是可以同时引入的，它们并不会相互影响，例如：

```
import myModule,{userName} from "module.js";
```

5-26　编程练习

练习1：分析下面代码的返回值。

```
(function(x, f = () => x) {
   var x;
   var y = x;
   x = 2;
   return [x, y, f()];
})(1)
```

解析：本题答案为[2,1,1]。本题有较大的迷惑性，考察的知识点也较多。首先考察了自执行匿名函数，传入了数值1作为参数，之后考察了ES6中函数的默认参数和箭头函数的相关知识，由于箭头的作用域固化为定义时的作用域，因此该箭头函数如果执行，将返回数值1。

之后执行y=x，因此y=1，接着执行x=2，因此返回结果为[2,1,1]。

练习2：分析下面代码的返回值。

```
(function() {
  return [
    (() => this.x).bind({ x: 'inner' })(),
    (() => this.x)()
  ]
}).call({ x: 'outer' });
```

解析：本题答案为['outer', 'outer']。其实本题并不复杂，主要考察箭头函数中this的固化，在箭头函数声明时，其中的this将固化成和当前作用于中的this一致，即外层函数的this，call方法将外层函数的this指定为对象{x: 'outer'}，因此两个箭头函数的执行结果均返回'outer'。

练习3：分析下面的代码执行后y的值。

```
let x, { x: y = 1 } = { x }; y;
```

解析：本题答案为“1”。本题主要考察解构赋值的相关知识，声明的x初始值为undefined，因此解构赋值失败会将y赋值为默认值1。

练习4：分析下面代码的结果。

```
[...[...'...']].length
```

解析：本题答案为“3”。本题主要考察扩展运算符“...”的使用，上面的代码将字符串进行扩展后，再将数组进行扩展，结果为长度为3的数组。

练习5：分析下面代码的输出结果。

```
const promise = new Promise((resolve, reject) => {
  console.log(1)
  resolve()
  console.log(2)
})
promise.then(() => {
  console.log(3)
})
console.log(4)
```

解析：上面的代码将输出1，2，4，3。本题主要考察Promise的用法，Promise是ES6中提供的一种异步编程解决方案，其then函数中的回调是异步执行的。

练习6：分析下面代码的打印结果。

```
Promise.resolve(1)
  .then((res) => {
    console.log(res)
    return 2
  })
  .catch((err) => {
    return 3
  })
  .then((res) => {
    console.log(res)
  })
```

解析：上面的代码将打印1，2。Promise可以进行链式调用，调用then或者catch函数后，都会重新返回一个新的Promise对象。

第 6 章

JavaScript 常用设计模式

从本章开始，我们将进入设计模式的学习。设计模式是不分语言的，甚至是不分行业的。简单地说，设计模式就是行业经验的积累所表现出的最佳实践。例如在盖房子时，人们最开始的尝试可能是一层一层地往上垒，后来发现对于高层建筑，这样的做法有很大缺陷，于是人们开始先搭框架再进行填充。这个过程就是经验的积累，最后形成的建筑流程方式就是设计模式。对于软件开发，解决一个问题的方式可能有很多种，原则上你可以不使用任何设计模式和任何开发技巧来完成一个大型项目的开发，暂且不说你能否这样完成一个项目，即便能，之后的维护和扩展工作也将十分恐怖。

其实，在前面的学习中，你已经接触过设计模式，只是我们没有系统地进行总结学习，例如使用工厂模式来模拟类、使用原型模式来实现继承、使用迭代器模式来实现枚举等。本章将通过实践来更深入地理解“设计模式”在开发中的应用。相信通过本章的学习，除了可以提高你的编程能力外，你思考问题的方式也会更加灵活。

6-1　工厂设计模式

工厂模式是开发中十分常用的一种设计模式，核心是将对象的组装过程封装在工厂内

部，对外提供统一的调用接口。举个现实中的例子，客户在购买汽车时并不需要知道汽车是如何生产的，也不需要知道各个品牌的汽车结构有什么不同，客户只需要关心如何操作汽车就可以了（几乎所有汽车的操作方式都没有什么不同）。在编程开发中，工厂模式主要由接口协议、实现类和工厂函数组成，如图6-1所示。

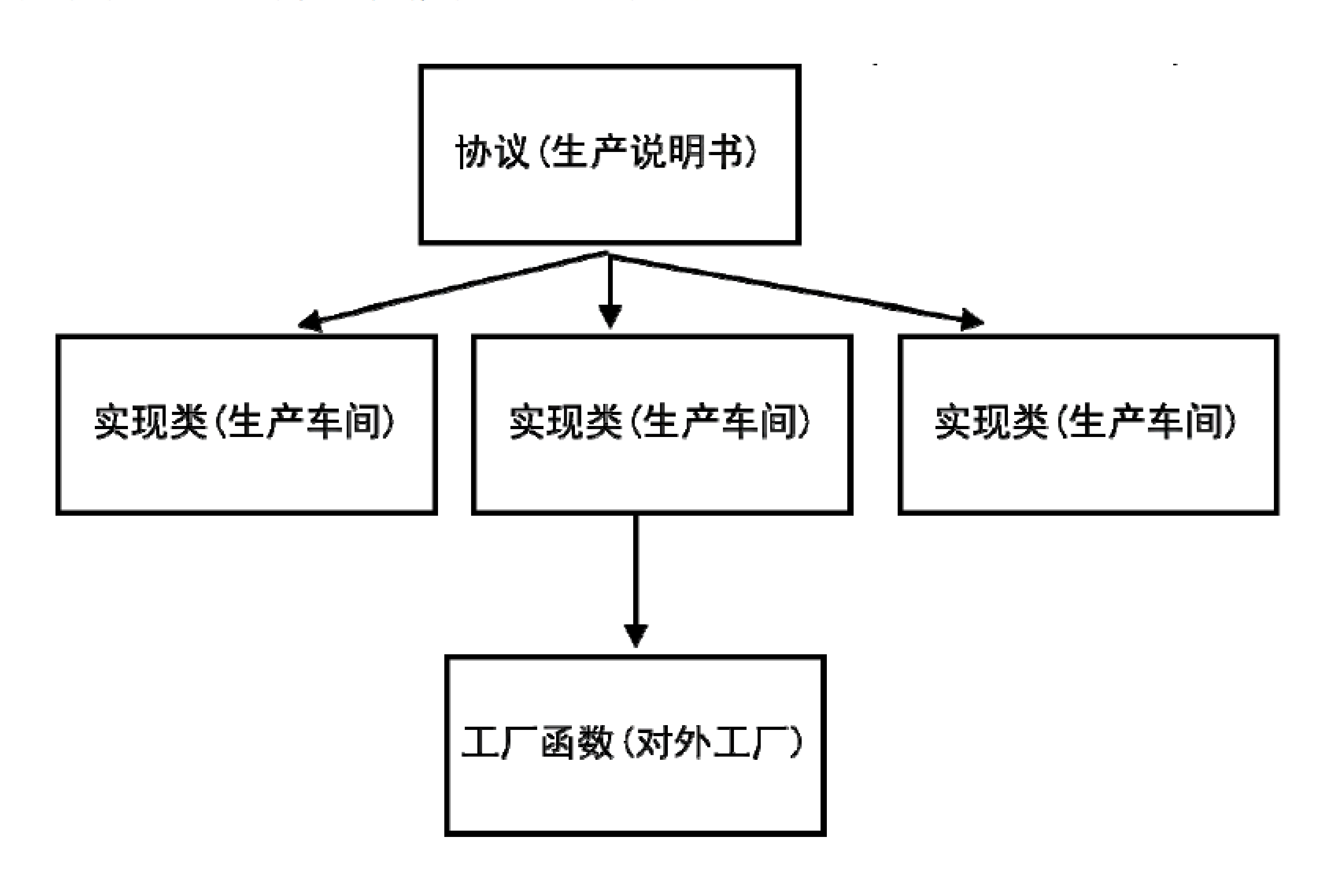


图 6-1　工厂模式示意图

在工厂模式中，接口用来定义对外协议，也可以简单理解为生产说明书，就像任何汽车都要按照指定的行驶标准生产；实现类是针对同一个协议的不同实现，可以对比理解为生产不同型号的汽车；工厂函数是直接供开发者调用的方法，用来进行对象实例的组装或创建。

针对实现工厂模式，许多编程语言都有先天的优势，例如Java中有接口概念，Swift中有协议概念。在JavaScript中，我们可以通过原型的方式来模拟协议。下面我们使用工厂模式实现一个图形工厂。

首先，你需要制定一个接口协议，例如所有图形都提供一个draw方法，这个方法的作用可能是将图形绘制在屏幕上，目前可以用打印的方式进行简单模拟。用一个shapeInterface对象来描述协议，代码如下：

```
var shapeInterface = {
    draw:function(){
        throw "must be implementation"
    }
}
```

上面的代码提供了一个draw方法，却在其中直接抛出了异常，这样做的目的是提醒所有实现这个协议的类都必须重写这个draw方法，如果不这样做，就失去了协议的约束作用（就像

要生产汽车，必须符合国家的标准规定）。我们再来创建两个图形类，以Circle和Rect为例：

```
function Circle(){
        this.draw=function(){
                console.log("Circle");
        }
}
Circle.prototype = shapeInterface;
function Rect(){
        this.draw=function(){
                console.log("Rect");
        }
}
Rect.prototype = shapeInterface;
```

实现一个图形工厂函数如下：

```
function Shape(type){
        if (type==="Circle") {
                return new Circle();
        }else if(type==="Rect"){
                return new Rect();
        }
}
```

在实际开发中，只会将shapeInterface和Shape函数暴露给其他开发者，会将Circle和Rect作为内部私有的类，这样做的好处是将对象的真实构建封装了起来，对于其他开发者来说，他们只需要传入自己想创建的图形类型，调用工厂函数来构建图形对象即可，例如：

```
var shape = Shape("Circle");
var shape2 = Shape("Rect");
shape.draw();           //Circle
shape2.draw();          //Rect
```

本章的学习方式一开始会让你觉得不太习惯，毕竟设计模式更多的是用在大型项目的框架设计中。本章中的示例也只是尽可能地将每种模式的结构和编写流程展示出来，可能你会觉得画蛇添足、去直求曲，但是只要静下心来思考这些设计模式的编程思路，一定会让你受益匪浅。

6-2　单例设计模式

单例模式是一种非常流行的设计模式。在介绍单例模式的概念前，我们先来看一个生活中的小例子。随着互联网深入人们的生活，现在越来越多的人选择在网上购买火车票，不知你有没有思考过这样的场景，每个人在网上购买车票时都会访问票务中心，由票务中心来负责车票余票的管理、车次信息的实时更新等。其实这个票务中心就起到了单例的作用，所有人访问到的票务中心都是同一个票务中心，只有这样才能保证每个人获取的信息是一致的，一个人购票成功后，所有人都会看到这列车的余票减少。

对于软件开发，单例模式的应用场景更加广泛，例如有登录功能的软件，用户信息的管理类通常用单例模式来设计，还有有关文件和数据的操作类，也通常使用单例来实现。单例设计模式符合如下3条规则：

（1）单例只能有一个实例。
（2）单例必须为其他所有调用方提供这一实例。
（3）单例实例一旦创建，就不会轻易销毁。

在JavaScript中，实现单例设计模式通常采用如下方式：

（1）定义一个全局对象，作为可以全局访问单例的接口。
（2）定义一个单例方法，用来获取单例实例。
（3）在使用单例时，若单例实例不存在，则进行创建；若存在，则直接返回。

示例代码如下：

```
var Singleton = {
    instance:function(){
        if (Singleton.__instance===undefined) {
            Singleton.__instance = {
                name:'关羽',
                weapon:'青龙偃月刀',
                blood:100,
                skill:function(){
                    console.log("挥刀斩");
                }
            }
        }
        return Singleton.__instance;
```

```
        }
    }
    var user = Singleton.instance();
    console.log(user.name,user.weapon);          //关羽  青龙偃月刀
    user.skill();        //挥刀斩
    user.skill=function(){
        console.log("斩首");
    }
    Singleton.instance().skill();        //斩首
```

6-3　建造者设计模式

建造者模式的核心是使用多个简单的对象构造复杂对象。在介绍设计模式的教程中，常常拿建别墅的例子来描述建造者。一个老板需要购买一套别墅房产，他首先会找到开发商，直接与开发商交涉来购买别墅，开发商会联系各个生产商或包工房，例如找来建筑工人盖房、泥瓦工人刷墙、装修工人进行室内装修等。当别墅建好后，开发商直接对老板进行交付。在这个场景中，老板就是我们编程中复杂对象的使用方，开发商就是建造者，而具体的建筑工人、泥瓦工人和装修工人等就是生产者。建造者将复杂对象的创建过程拆分成一个个简单的对象交给相应的生产者完成创建，最终建造者将创建出的简单对象进行组合，以提供复杂对象供使用者使用，如图6-2所示。

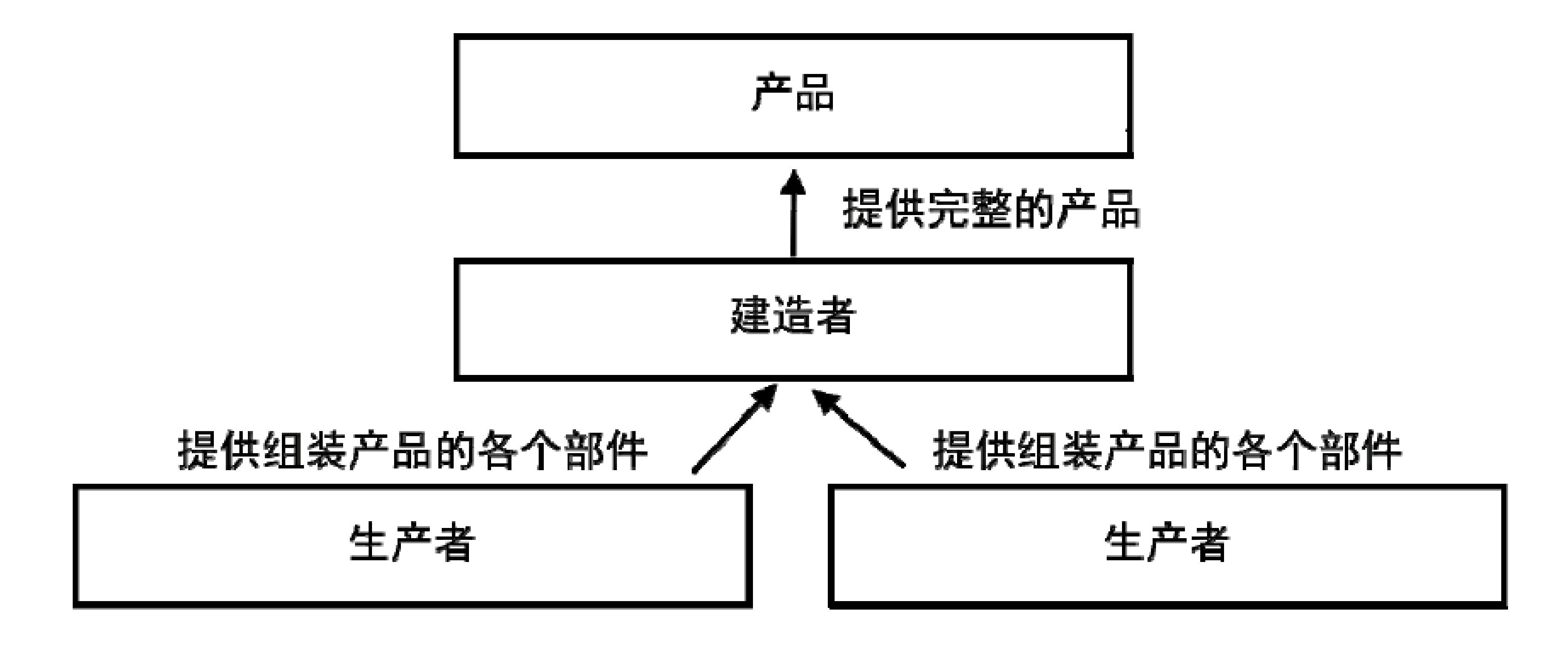


图 6-2　建造者模式示意图

建造者模式有如下优点：

（1）建造者只独立完成拼装过程，易扩展、易维护。

（2）将复杂对象进行拆分，隐藏对象构建的内部细节，符合单一功能原则。

下面换一个例子，使用JavaScript来模拟一个外卖套餐的建造者模式实现。首先，通过如下3个步骤来构建建造者模式。

（1）制定协议。
（2）拆分生产者。
（3）建造者进行组合。

协议的目的是为了提供给使用者使用接口，也是为了约束建造者构建的对象必须遵守约定。代码如下：

```
var mealInterface = {
    getName:function(){
        throw "必须有套餐名";
    },
    getStaple:function(){
        throw "必须有主食";
    },
    getDrink:function(){
        throw "必须有饮料";
    }
}
```

生产者用来创建被查分后的具体对象，比如饮料和主食：

```
function Drink(type){
    switch(type){
        case 1:{
            return "可口可乐";
        }
        break;
        case 2:{
            return "热牛奶";
        }
        break;
    }
}
function Staple(type){
    switch(type){
        case 1:{
            return "汉堡包";
        }
```

```
            break;
            case 2:{
                return "鸡肉卷";
            }
            break;
        }
}
```

下面我们以餐厅作为建造者进行套餐的组合：

```
function restaurant(type){
    switch(type){
        case 1:{
            let food = Object.create(mealInterface);
            food.getName = function(){
                return "可乐汉堡套餐";
            }
            food.getDrink = function(){
                return Drink(type);
            }
            food.getStaple = function(){
                return Staple(type);
            }
            return food;
        }
        break;
        case 2:{
            let food = Object.create(mealInterface);
            food.getName = function(){
                return "鸡肉卷牛奶套餐";
            }
            food.getDrink = function(){
                return Drink(type);
            }
            food.getStaple = function(){
                return Staple(type);
            }
            return food;
        }
```

```
                break;
        }
}
```

使用如下：

```
var f = restaurant(2);
console.log(f.getName(),f.getStaple(),f.getDrink());        //鸡肉卷牛奶套餐  鸡肉卷  热牛奶
```

6-4　适配器设计模式

适配器模式虽然也被称为设计模式，其实在很多情况下，适配器模式都不是使用在程序结构设计中，而是为了应用而临时采取的一种妥协方案。适配器模式为两个互不兼容的系统提供可以调用的接口。举个生活中的常见场景，现在我们周围到处都被电子设备充斥着，各种设备在充电时需要的电压可能都不同，但是电厂提供给用户的都是统一的220V交流电，这就需要使用变压器来将统一的220V电压转换成各种设备需要的电压。适配器就起到这样一种作用。

下面我们来看JavaScript中如何应用适配器模式。例如，有一个已经编程完成的用于数组排序的对象，代码如下：

```
var sort = {
        type:"DESC",
        data:[],
        run:function(){
                if (this.type==="DESC") {
                        return this.data.sort(function(a,b){
                                return a<b;
                        })
                }
                if (this.type==="ASC") {
                        return this.data.sort(function(a,b){
                                return a>b;
                        })
                }
        }
}
```

可以使用如下方式进行使用：

```
sort.type = "ASC";
sort.data = [2,3,1,5,3,2,6];
console.log(sort.run());      //[ 1, 2, 2, 3, 3, 5, 6 ]
```

现在假如有一个用户输入系统，只能接受如下格式的排序命令：

```
"D 1,2,3,5,1,2"        //表示对后面的数值进行降序并返回数组
"A 3,1,2,4,2,1,4"      //表示对后面的数值进行升序并返回数组
```

虽然sort对象可以完成排序逻辑，但是其不能识别这种字符串形式的输入，这时就需要使用适配器来对sort对象进行转换，代码如下：

```
var d1 = "D 1,2,3,5,1,2"          //表示对后面的数值进行降序并返回数组
var d2 = "A 3,1,2,4,2,1,4"        //表示对后面的数值进行升序并返回数组
var sortAdapt = {
    data:"",
    run:function(){
        let array = this.data.split(" ");
        let type = array[0];
        sort.type = type==="D"?"DESC":"ASC";
        sort.data = array[1].split(",");
        return sort.run();
    }
}
sortAdapt.data = d2;
console.log(sortAdapt.run());          //[ '1', '1', '2', '2', '3', '4', '4' ]
```

sortAdapt就是sort对象的适配器，其属性和方法都与sort基本保持一致，并且满足了这种输入设备的命令格式。

6-5　装饰器设计模式

装饰器模式的作用是扩展已有类的新功能。装饰器模式也可以理解为对已有类的一种包装，在保持原类结构不修改的前提下扩展类的功能。例如，游戏中的武将有一些物品和技能，同样武将也可以骑马，我们可以通过装饰器模式来对武将对象进行装饰，让其有骑马冲锋的能力。示例代码如下：

```
var general = {
    name:"关羽",
```

```
        weap:'青龙偃月刀',
        skill:function(){
            console.log(this.name+" "+this.weap+" "+"挥刀斩");
        }
    }
    var GeneralPack = function(general){
        this.name = general.name;
        this.weap = general.weap;
        this.horse = "赤兔马";
        this.skill = function(){
            console.log(this.name+" "+this.weap+" "+this.horse+" "+"挥刀斩");
        }
    }
    general.skill();        //关羽 青龙偃月刀 挥刀斩
    var general_2 = new GeneralPack(general);
    general_2.skill();      //关羽 青龙偃月刀 赤兔马 挥刀斩
```

上面的代码中，general对象是武将对象，GeneralPack函数是一个装饰器类，用来构造武将对象经过装饰器装饰后的对象。需要注意，一般我们会保持装饰后的对象和原对象同样的接口，这样对装饰器的使用者来说，这个装饰对象实际上是透明的，只是做了武将能力的扩展。

6-6　外观设计模式

外观设计模式有时也称门面模式，它的核心其实也是组合与包装。外观模式可以将复杂的子系统进行组合，对外提供统一的访问接口，隐藏系统内部的复杂结构。例如，我们去医院看病时，可能要经历挂号、问诊、检查、开药、付费、取药这些过程。如果第一次去这个医院，不熟悉流程将会耗费大量时间。这时，如果有一个指示牌，将上述过程与服务地点清晰地标识出来，将大大提高我们看病的效率。这个指示牌就充当了外观设计模式中的门面对象。下面用JavaScript代码来模拟上面的过程。

```
    var register = {
        wait:function(){
            console.log("排队等待取号");
        },
        regist:function(){
            console.log("拿到挂号");
```

```
        }
}
var doctor = {
        diagnose:function(){
                console.log("进行问诊");
        },
        watchCheck:function(check){
                console.log("查看检查结果："+check);
        },
        medication:function(){
                console.log("开发药方");
                return "药方 A";
        }
}

var check = {
        makeCheck:function(){
                console.log("做检查");
                return "检查结果";
        }
}
var medication = {
        cost:function(medication){
                console.log("付费："+medication);
        },
        take:function(){
                console.log("取药回家");
        }
}
/*
排队等待取号
拿到挂号
进行问诊
做检查
查看检查结果：检查结果
开发药方
付费：药方 A
取药回家
*/
```

```
register.wait();
register.regist();
doctor.diagnose();
var res = check.makeCheck();
doctor.watchCheck(res);
var me = doctor.medication();
medication.cost(me);
medication.take();
```

可以看到，要完成这样一个看病流程，开发者需要调用多个对象的多个方法，并且需要进行参数的来回传递，我们可以使用外观设计模式来简化这一过程的调用。

```
var hospital = {
    watch:function(){
        register.wait();
        register.regist();
        doctor.diagnose();
        var res = check.makeCheck();
        doctor.watchCheck(res);
        var me = doctor.medication();
        medication.cost(me);
        medication.take();
    }
}
hospital.watch();
```

外观模式十分适用于这样的场景：多个方法组合调用，并且参数传递只在这些方法内部使用。使用外观模式不仅可以简化调用过程、降低错误率，也可以提高代码的复用性。

6-7　享元设计模式

享元设计模式是一种对系统进行优化的设计模式，对于存在大量对象的系统，使用享元模式可以显著减轻内存负担。比如一个图书管理系统，其中要存放图书馆中所有图书的相关信息和借书相关信息，比如书号、书名、简介、分类、存放位置、借书人、借书时间等。一般来说，一座图书馆中的书成千上万，随着录入系统的图书越来越多，所消耗的内存也将越来越大。享元模式的核心是将对象通用且静态的部分与独立且动态的部分进行分离，使其内部状态和外部状态隔离开，将内部状态进行共享复用。

下面的代码模拟一个图书类，每当有图书录入系统时，都需要构建出这样一个类对象。

```
function Book(sn,name,content,author,reader,initTime,expriseTime){
    this.sn = sn;
    this.name = name;
    this.content = content;
    this.author = author;
    this.reader = reader;
    this.initTime = initTime;
    this.expriseTime = expriseTime;;
    this.show = function(){
        console.log(this.sn,this.name,this.content,this.author,this.reader,this.initTime,
this.expriseTime);
    }
}
var book = new Book(10011, "老人与海", "在逆境中坚持", "海明威", "Jaki", "2017.11.11",
"2017.12.12");
book.show();        //10011 '老人与海' '在逆境中坚持' '海明威' 'Jaki' '2017.11.11' '2017.12.12'
```

下面使用享元模式来修改上面的代码：

```
function Book_In(sn,name,content,author){
    this.sn = sn;
    this.name = name;
    this.content = content;
    this.author = author;
}

var bookPool = new Set();
function Book(sn,name,content,author,reader,initTime,expriseTime){
    for(var b of bookPool){
        if (b.sn===sn) {
            this.in = b;
        }
    }
    if (this.in==undefined) {
        this.in = new Book_In(sn,name, content, author);
        bookPool.add(this.in);
    }
    this.reader = reader;
```

```
            this.initTime = initTime;
            this.expriseTime = expriseTime;
    }
    Book.prototype = {
            show:function(){
                    console.log(this.in.sn,this.in.name,this.in.content,this.in.author,this.reader,
this.initTime,this.expriseTime);
            }
    }
    var book = new Book(10011,"老人与海","在逆境中坚持","海明威","Jaki","2017.11.11",
"2017.12.12");
    book.show();
    var book2 = new Book(10011,"老人与海","在逆境中坚持","海明威","Jaki","2017.11.11",
"2017.12.12");
    book2.show();
    console.log(book.in===book2.in);        //true
```

上面的代码将通用属性提取成享元，放入一个共享池中，在向系统中录入图书信息时，会优先从共享池中取数据（同一图书的SN编码都是唯一且一致的），如果没有，才进行享元的创建。从打印信息也可以看出，不同的图书录入对象实际上是共享同一个享元对象，享元设计模式可以大大减少内存的开销。

6-8　代理设计模式

在开发中，代理设计模式是一种十分实用的设计模式。其实在生活中，代理模式也无处不在，例如现在十分流行的网购，在买家和卖家之间的物流公司就可以理解为一个代理，当买家购买物品后，由物流公司代替卖家进行送货。同样在房屋租赁过程中，房产中介也可以理解为代理。下面用JavaScript代码来模拟一下网购中的代理模式。

```
    var me = {
            shop:undefined,
            buy:function(){
                    console.log("网购 "+this.shop.delever());
            }
    }
    var computerShop = {
            delegate:undefined,
```

```
        delever:function(){
            return this.delegate.delever();
        }
    }
    var logistics = {
        delever:function(){
            return "顺丰物流正在送货";
        }
    }
    me.shop = computerShop;
    computerShop.delegate = logistics;
    me.buy();        //网购 顺丰物流正在送货
```

如上面的代码所示，对买家来说，物品的运输是由卖方负责的，对卖方来说，他并不会自己去送货，即computerShop对象并不会自己去实现delever方法，而是交由一个代理对象实现，这样代理对象就和卖家对象进行解耦，并且买家、卖家、代理三方都可以灵活组合，例如买家可以选择其他的网店对象，只要其有delever方法，同样，网店也可以灵活搭配物流商，由具体的代理对象来完成送货功能。

6-9 责任链设计模式

责任链设计模式用来连接处理任务的一连串节点，使用责任链设计模式可以将复杂的处理过程拆分成多个负责方，并且可以灵活替换和修改。生活中的很多单位都采用这样的设计模式来组织结构，例如要开发一处房产，可能需要多个政府部门的批准与检查，每个部门对自己所管辖的范围负责。

在JavaScript代码中，我们可以使用责任链设计模式来实现校验网址功能。例如，当用户输入一个网址后，需要根据用户输入的信息进行网址的有效性判断，然后分别做出提示。

```
    var typeCheck = {
        nextHandler:undefined,
        handle:function(obj){
            if (typeof obj !=="string") {
                console.log("必须为字符串");
                return false;
            }
            if (this.nextHandler!=undefined) {
                this.nextHandler.handle(obj);
```

```
                }else{
                        console.log("校验成功");
                        return true;
                }
        }
}

var emptyCheck = {
        nextHandler:undefined,
        handle:function(obj){
                if (obj.length===0) {
                        console.log("字符串不能为空");
                        return false;
                }
                if (this.nextHandler!=undefined) {
                        this.nextHandler.handle(obj);
                }else{
                        console.log("校验成功");
                        return true;
                }
        }
}
var vaildCheck = {
        nextHandler:undefined,
        handle:function(obj){
                if (!/^(www\.).+(\.com)$/.test(obj)) {
                        console.log("字符串不合规");
                        return false;
                }
                if (this.nextHandler!=undefined) {
                        this.nextHandler.handle(obj);
                }else{
                        console.log("校验成功");
                        return true;
                }
        }
}
typeCheck.nextHandler = emptyCheck;
emptyCheck.nextHandler = vaildCheck;
typeCheck.handle("www.ws.com");
```

上面的代码创建了责任链中的3个节点，第1个节点用来校验传入的数据是否为字符串类型，第2个节点用来校验传入的字符串是否为空字符串，第3个节点用来进行网址的正则匹配。任何一个阶段校验失败都会中断后面的校验。这种责任链将十分容易扩展，如果需要将校验规则改成邮箱，只需要在vaildCheck后面再添加一个邮箱校验的节点即可。

6-10　命令设计模式

命令设计模式常常用来进行复杂功能实现与使用的解耦。其核心是将功能的调用封装成命令，对使用方来说，只需要提供正确的命令和参数即可。如果你在电脑上使用过命令行工具，理解命令设计模式将十分容易。下面使用JavaScript代码来模拟使用命令模式进行四则运算。

```
var Calu = {
    perform:function(p){
        this.run(p.type,p.params);
    },
    run:function(command,params){
        this[command](...params);
    },
    add:function(a,b){
        console.log("a+b=",a+b);
    },
    sub:function(a,b){
        console.log("a-b=",a-b);
    },
    mul:function(a,b){
        console.log("a*b=",a*b);
    },
    div:function(a,b){
        console.log("a/b=",a/b);
    }
}
var p = {
    type:"add",
    params:[10,5]
}
var p2 = {
    type:"sub",
```

```
        params:[10,5]
    }
    var p3 = {
        type:"mul",
        params:[10,5]
    }
    var p4 = {
        type:"div",
        params:[10,5]
    }
    Calu.perform(p);
    Calu.perform(p2);
    Calu.perform(p3);
    Calu.perform(p4);
```

如上面的代码所示，Calu是核心的四则运算对象，其中提供了加减乘除等函数，对于使用方来说，我们并不直接调用这些函数，要进行四则运算，需要通过创建一个命令对象，这个对象的type属性用来指定要进行的运算类型，params参数用来设计进行运算的数值，通过run函数对命令的解析来执行具体的功能函数，这样就完成了使用和实现的解耦。这种设计模式在实际开发中也有广泛的应用，例如在使用Ajax进行网络请求时，许多第三方的封装都会采用命令模式来进行请求的配置。

6-11　迭代器设计模式

关于迭代器，你其实并不陌生，在前面的学习中也使用过迭代器为自定义对象扩展支持for-of遍历。JavaScript中内置实现的迭代器模式前面只是使用，并没有详细思考它的设计原理。迭代器设计模式可以统一遍历集合对象的方式，并且十分易于扩展，开发者可以十分容易地让自定义的集合支持统一提供的迭代器。并且，对于使用方来说，统一的接口可以更加便于进行代码的组织与管理。下面我们使用JavaScript代码来模拟实现一个简单的迭代器。

```
    //定义一个迭代器协议
    var MyIterator = {
        next:function(){
            throw "必须实现 next"
        }
    }
    //迭代器方法
```

```
function tool(obj,callback){
        var item = obj.next();
        while(item!==undefined){
            callback(item);
            item = obj.next();
        }
}
//实现迭代器的对象
var myObj = {
    name:'Jaki',
    age:24,
    subject:"JavaScript",
    teaching:function(){
        console.log("teaching");
    },
    _keys:["name","age","subject"],
    _tip:0,
    next:function(){
        return this._keys[this._tip++];
    }
}
myObj.__proto__=MyIterator;
//验证
tool(myObj,function(item){
    console.log(item);
});
```

上面的示例代码就是迭代器设计模式的工作思路，当有新的集合对象需要支持迭代时，开发者只需要在其中实现next方法，tool函数就可以很好地对其进行工作。

6-12　备忘录设计模式

备忘录设计模式也可以理解为一种备份设计模式，用来进行对象内部状态的备份。当然，备份的可能并不是完整的对象，我们在使用文本编辑器时可以撤销，浏览网页时可以返回，都是备忘录模式的实际应用。下面我们用JavaScript来演示关于用户输入与撤销的备忘录模式的实现。

```
var EditProtocol = {
    _stateArray:[],
    revoke:function(){
        this._refresh(this._stateArray.pop());
    },
    save:function(){
        this._stateArray.push(this._state());
    },
    _refresh:function(){
        throw "必须实现_refresh 方法";
    },
    _state:function(){
        throw "必须实现 state 方法";
    }
}
var teacher = {
    name:"Jaki",
    age:25,
    subject:"JavaScript",
    _refresh:function(state){
        this.name = state.name;
        this.age = state.age;
        this.subject = state.subject;
    },
    _state:function(){
        return {
            name:this.name,
            age:this.age,
            subject:this.subject
        }
    }
}

teacher.__proto__ = EditProtocol;
teacher.save();
console.log(teacher.name,teacher.age,teacher.subject);          //Jaki 25 JavaScript
teacher.name = "Lucy";
teacher.age = 23;
console.log(teacher.name,teacher.age,teacher.subject);          //Lucy 23 JavaScript
```

```
teacher.revoke();
console.log(teacher.name,teacher.age,teacher.subject);          //Jaki 25 JavaScript
```

上面的代码中，我们使教师对象支持撤销操作，备忘录模式核心的协议是EditProtocol对象，任何需要支持撤销操作的对象，只要以它为原型并且实现了必须重写的方法，就可以完成状态的保存与恢复。备忘录模式在开发中的实际应用价值很大，除了在状态恢复的场景中需要外，有时候也可以用来缓存网络数据。

6-13　观察者设计模式

观察者设计模式常常用来实现订阅逻辑。当对象间出现一对多的交互关系时，也可以使用观察者模式来处理。例如在现实生活中，杂志社与读者间就是观察者设计模式的实际应用。观察者设计模式一般由服务端和客户端两方组成，客户端通过订阅来监听服务端的信息，由服务端发送信息给已经订阅的客户端。下面我们使用JavaScript通过观察者来实现如下逻辑：教务处作为服务端，会给教师发布任务，当教师接收到任务后，开始执行教学任务，代码如下：

```
var Service = {
    _customer:new Set(),
    addObserver:function(obj,func){
        this._customer.add({obj:obj,func:func});
    },
    deleteObserver:function(obj){
        var has;
        this._customer.forEach(function(object){
            if (object.obj===obj) {
                has = object;
            }
        });
        if (has!==undefined) {
            this._customer.delete(has);
        }
    },
    publish:function(msg){
        console.log("发布了订阅消息");
        this._customer.forEach(function(obj){
            obj.func.call(obj.obj,msg);
        });
```

```
        }
    }

    var teacher1 = {
        name:"Jaki",
        teach:function(msg){
            console.log(this.name+"开始教学"+msg);
        }
    }
    var teacher2 = {
        name:"Lucy",
        teach:function(msg){
            console.log(this.name+"开始教学"+msg);
        }
    }
    Service.addObserver(teacher1,teacher1.teach);
    Service.addObserver(teacher2,teacher2.teach);
    /*
    发布了订阅消息
    Jaki 开始教学 Swift
    Lucy 开始教学 Swift
    */
    Service.publish("Swift");
    Service.deleteObserver(teacher1);
    /*
    发布了订阅消息
    Lucy 开始教学 JavaScript
    */
    Service.publish("JavaScript");
```

如上面的代码所示，我们为服务端提供了添加订阅与删除订阅的方法，只要服务端有发布动作，所有观察者都可以监听到并执行预定的任务。

6-14　编程练习

练习1：试着使用工厂设计模式来模拟一个汽车工厂。

解析：可参考如下代码：

```
//协议接口
var CarInterface = {
        des:function(){
                throw "must be have a function 'des'";
        }
}
function Car(){
        this.des = function(){
                console.log("我是一辆轿车");
        }
}
Car.prototype = CarInterface;
function Bus(){
        this.des = function(){
                console.log("我是一辆公交车");
        }
}
Bus.prototype = CarInterface;
//工厂函数
function CarFactory(type){
        if (type=='Car') {
                return new Car();
        }else{
                return new Bus();
        }
}
var obj = CarFactory('Car');
obj.des();          //我是一辆轿车
```

工厂模式的优势是：当我们创建某一个对象时，可以不用关心其具体的类。

练习2：试着用观察者模式来实现如下逻辑：

有3个对象：猫、老鼠和人。猫看见了老鼠，老鼠吓跑了，人被吵醒。

解析：可参考如下代码：

```
var cat = {
        _outer:new Set(),
        addObserver:function(obj,callback){
                this._outer.add({obj:obj,callback:callback});
        },
```

```
        deleteObserver:function(obj){
            var has;
            this._outer.forEach(function(object){
                if (object.obj===obj) {
                    has = object;
                }
            });
            if (has!==undefined) {
                this._outer.delete(has);
            }
        },

        shout:function(){
            console.log("喵");
            this._outer.forEach(function(obj){
                obj.callback.call(obj.obj);
            });
        }
    }

    var mouse = {
        run:function(){
            console.log("老鼠吓跑了");
        }
    }

    var man = {
        wake:function(){
            console.log("人被惊醒了");
        }
    }

    cat.addObserver(mouse,mouse.run);
    cat.addObserver(man,man.wake);
    cat.shout();
    //将打印
    //喵
    //老鼠吓跑了
    //人被惊醒了
```

第 7 章

JavaScript HTML DOM/BOM

DOM的全称是Document Object Model（文档对象模型），它是HTML的标准对象模型，也是HTML的标准编程接口。简单来说，它提供了JavaScript对HTML文档内容进行获取、修改、添加或删除的功能。网页的展现完全依靠HTML文档，因此也可以理解为DOM提供了用JavaScript动态修改网页的功能。

BOM是指Browser Object Model，即浏览器对象模型。DOM是文档模型，JavaScript HTML DOM是提供给开发者操作文档的接口，同样，JavaScript HTML BOM是提供给开发者操作浏览器窗口的接口。BOM主要包含5个对象：Window、Navigator、Screen、History和Location。这些对象各有用途，开发者通过它们可以获取窗口位置尺寸、跳转历史、URL信息等各种浏览器信息。

本章主要介绍DOM的使用及BOM相关的5个对象中提供的常用属性与方法。

7-1　创建学习模板

本章开始，我们编写的代码将不再只是逻辑上的演示，还会有许多界面元素。因此，我们将使用网页浏览器作为学习的工具，虽然任何浏览器都可以运行我们编写的代码，但是建

议安装一个Google Chrome浏览器，它有强大的开发者功能，可以帮助你调试与检查代码。首先在桌面上新建一个文件夹，将其命名为project，在其中再创建两个文件夹和一个index.html文件，文件夹分别命名为demo与javaScript。使用Sublime Text工具直接打开桌面上的project文件夹（将文件夹直接拖入Sublime Text工具即可），你的项目结构看起来应该是如图7-1所示的样子。

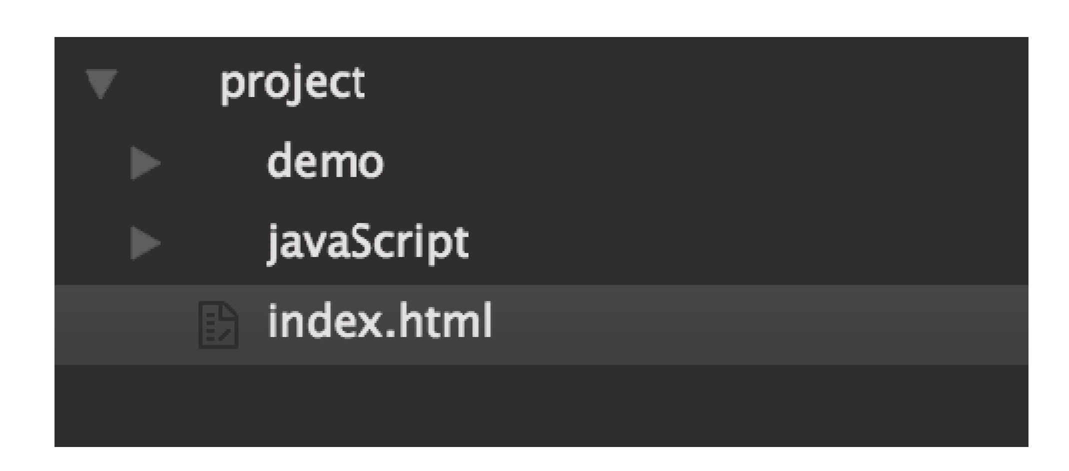


图 7-1 工程模板结构

在之后的学习中，我们会将每个示例HTML文件放入demo文件夹中，将每个示例JavaScript文件放入javaScript文件夹中，index.html文件作为目录列表，这样可以更加清晰地管理每个示例文件，即使时间过去很久，你需要查阅相关内容时，也会一目了然。举个例子，先编写index.html文件如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>示例</title>
</head>
<body style="margin: 100px">
<a href="./demo/demo.html">示例一</a>
</body>
</html>
```

在demo文件夹下新建一个demo.html文件，编写如下代码：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>框架示例</title>
</head>
<body>
<h1>我是程序框架示例</h1>
```

```
</body>
</html>
```

之后直接在浏览器中打开index.html，单击“示例一”标题，即可跳转到demo.html页面。好了，我们的准备工作做完了，愉快地开始本章的学习吧。

7-2　几个重要概念

HTML DOM是树状结构，其中所有的事物都是节点。根据标准定义，整个文档是一个文档节点，每个HTML元素是元素节点，元素内的文本是文本节点，每个HTML元素的属性是属性节点，注释是注释节点。图7-2是一个HTML DOM树的示例。

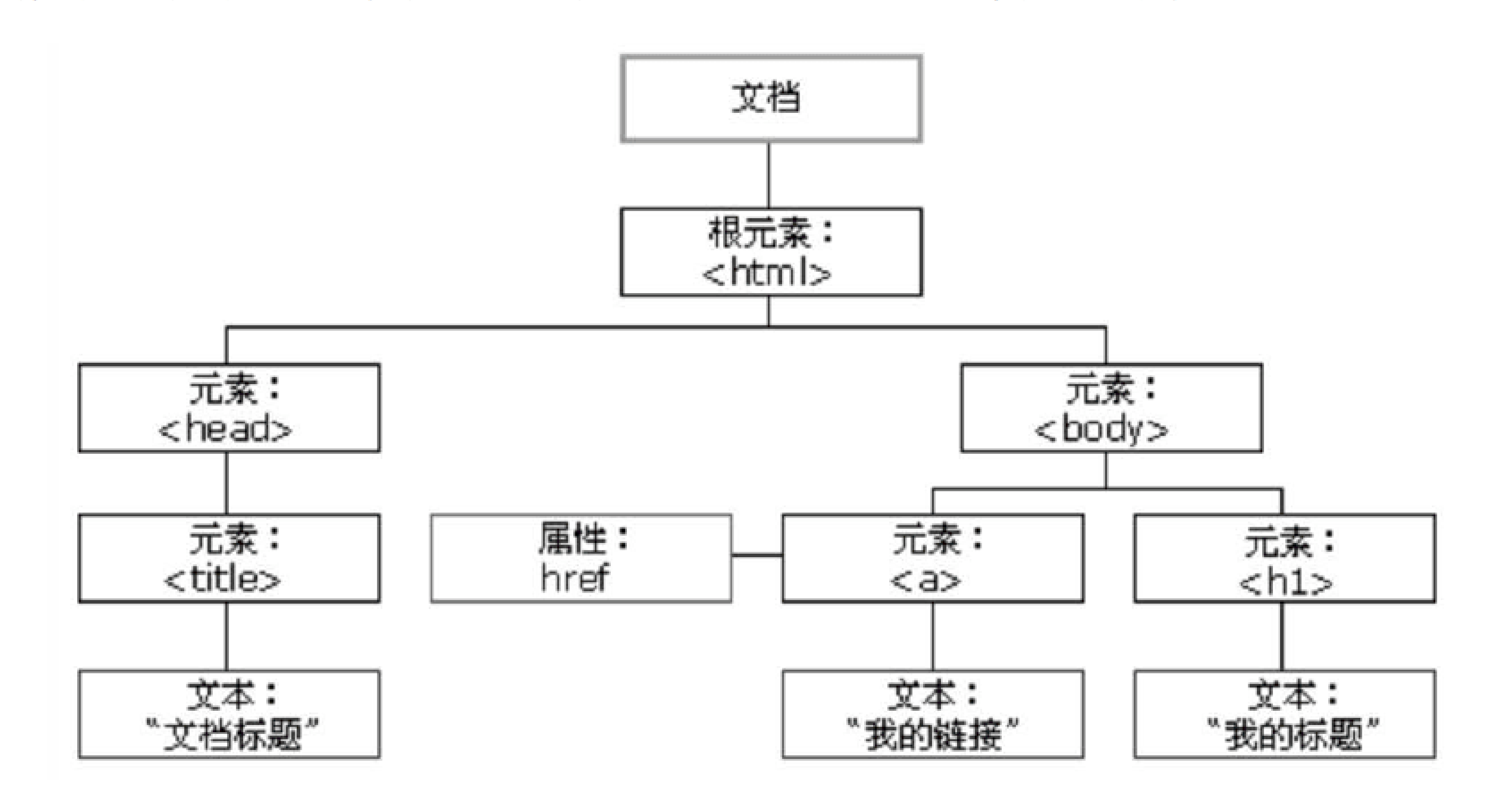


图 7-2　HTML DOM 树示例

既然是树结构，就存在一些层级关系。在节点树中，顶端节点被称为根节点，例如图7-2中的<html>节点。除了根节点外，每个节点都有父节点，例如图7-2中的<body>和<head>节点的父节点是<html>节点，<html>节点的子节点是<head>节点和<body>节点。拥有相同父节点的节点互为同胞节点，例如图7-2中的<head>节点和<body>节点互为同胞节点。需要注意，文本也是一种节点（文本节点），例如图7-2中的“我的链接”就是<a>节点的子节点。

7-3　Document 文档对象

每个载入浏览器的文档都会成为Document对象，Document对象为开发者提供了对HTML页面中所有元素的访问接口，在前面创建的project工程的demo文件夹下新建一个命名

为documentDemo.html的文件，在javaScript文件夹下新建一个命名为documentJs.js的文件。编写documentDemo.html文件如下：

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="utf-8">
	<title>Decument</title>
</head>
<body>
<h1>标题</h1>
<p>段落一段落一段落一段落一段落一段落一段落一段落一段落一段落一</p>
<p>段落二段落二段落二段落二段落二段落二段落二段落二段落二段落二段落二</p>
<a name="first">第一个锚</a><br />
<a name="second">第二个锚</a><br />
<a name="third">第三个锚</a><br />
<script type="text/javascript" src="../javaScript/documentJs.js"></script>
</body>
</html>
```

编写documentJs.js文件如下：

```
(function(){
	console.log("文档所包含的所有节点");
	for (var i = document.all.length - 1; i >= 0; i--) {
		var element = document.all[i];
		console.log(element.localName);
	}
	console.log("文档中锚点数："+document.anchors.length);
})();
```

修改index.html文件如下，添加示例入口：

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="utf-8">
	<title>示例</title>
</head>
<body style="margin: 100px">
<a href="./demo/demo.html">示例一</a>
```

```
<a href="./demo/documentDemo.html">Decument 示例</a>
</body>
</html>
```

在浏览器中运行，打开调试模式（Chrome浏览器可以使用Option+Command+J来打开调试模式），观察打印区，可以看到文档中所有节点的名称。

document对象的all属性用来获取文档中所有的HTML节点对象，这个属性是一个线性列表，anchors属性则用于获取文档中的所有锚点对象。除了这两个属性外，document对象还有4个列表属性，分别是applets、forms、images、links。applets获取文档中的所有Applet对象，forms获取文档中的所有表单对象，images获取文档中的所有Area和Link对象。document对象定义的其他常用属性如表7-1所示 。

表7-1　document对象定义的其他常用属性

属性名	意　义
Body	获取文档中的 body 节点对象
Cookie	用来设置和获取 Cookie
Domain	获取载入当前文档的 Web 主机名
lastModified	获取当前文档的最后修改日期
Referrer	获取载入当前文档的 URL，必须是从超链接载入这个属性才有值
Title	获取当前文档的标题
URL	获取当前文档的 URL

需要注意，cookie属性在设置浏览器Cookie时，使用如下格式进行设置：

```
document.cookie = "userName=jaki";
```

设置的字段会被追加到现有的Cookie中去（Chrome浏览器不支持静态文件设置Cookie，测试时需注意使用其他浏览器）。

document对象中定义的常用方法如表7-2所示。

表7-2　document对象定义的常用方法

方法名	参　数	意　义
Open	(mimetype,replace) mimetype 用来指定文档类型，默认为"text/html" replace 只有一个选项"replace"，这个参数是否提供决定了新打开的文档是否替换父文档	打开一个新的文档，将原内容擦除

（续表）

方法名	参　数	意　义
Write	(exp1,exp2...) 参数将按顺序写入文档	向文档中写入内容
Writeln	(exp1,exp2...) 参数将按顺序写入文档	作用同 write 函数，不同的是在每次写入后都会加换行
Close	无	关闭使用 open 方法打开的文档
getElementById	(id)	获取根据指定 id 查找到的第一个节点对象
getElementsByName	(name)	获取根据 name 属性查找到的节点对象集合
getElementsByTagName	(tag)	获取根据 HTML 标签名获取到的节点对象集合

上面列举的方法中，3个获取节点的方法在实际开发中十分常用，主要通过它们来实现对网页的动态更改。

7-4　Element 节点对象

上一示例中我们了解到，在一个HTML文档中有4种节点，HTML标签是元素节点，标签属性为属性节点，标签内的文本为文本节点，注释为注释节点。JavaScript可以操作具体的节点Element对象来对节点进行增、删、改、查等操作。请看如下示例代码：

HTML文件：

```
<!DOCTYPE html>
<html>
<head>
        <title>ElementDemo</title>
</head>
<body>
<a id="a" href="https://www.baidu.com">链接 1</a><p id='p'>文本</p><p id="p">文本</p>
<script type="text/javascript" src="../javaScript/elementJs.js"></script>
</body>
</html>
```

JavaScript示例代码：

```
        let element = document.getElementById("a");
```

```
let newElement = document.createElement("a");
newElement.innerHTML = "链接 2";
element.appendChild(newElement);
```

使用getElementById函数返回的对象就是节点对象，可以调用document对象的createElement方法来创建新标签节点，这个函数中需要传入HTML标签名作为参数。Element对象的appendChild方法用来在当前节点内追加子节点，调用此函数后，效果会直接展示在浏览器界面上。

Element对象上也定义了许多常用属性，如表7-3所示。

表7-3　Element对象定义的常用属性

属性名	意　义
childNodes	获取节点中所有子节点，需要注意，元素中的文本也是节点，也会被获取到
attributes	获取节点的属性集合
className	获取节点的 class 属性值
clientHeight	获取元素的可见高度
clientWidth	获取元素的可见宽度
contentEditable	设置或获取元素是否可编辑，可以设置为 true、false 和 inherit，设置 inherit 表示默认与父元素一致
dir	设置或获取元素的文本方向
firstChild	获取元素节点下的第一个子节点（可以是文本节点）
id	设置或获取当前节点 id
innerHTML	设置或获取元素的内容，以字符串的方式
lang	设置或返回标签的 lang 属性值
lastChild	获取节点内的最后一个子节点
nextSibling	获取当前节点的下一个同胞节点，需要注意，如果在 HTML 文件中两个节点间有空格，这个属性将获取到 undefined
nodeName	获取节点名称，如果是标签元素，就会获取标签名；如果是属性元素，就会获取属性名
nodeType	获取节点类型
nodeValue	获取节点的值
offsetHeight	获取元素高度
offsetWidth	获取元素宽度
offsetLeft	获取元素的水平偏移位置
offsetParent	获取元素计算偏移位置所参照的父节点
offsetTop	获取元素的垂直偏移位置
ownerDocument	获取根节点 Document 对象

（续表）

属性名	意　义
parentNode	获取元素的父节点
previousSibling	获取节点的前一个同胞节点
scrollHeight	获取元素的整体内容高度
scrollWidth	获取元素的整体内容宽度
scrollTop	元素的整体内容高度减去元素的可见高度
scrollLeft	元素的整体内容宽度减去元素的可见宽度
style	设置或获取元素的 style 属性节点
tabIndex	设置或获取元素的 tab 切换顺序
tagName	获取元素的标签名
textContent	设置或获取元素的文本内容
title	设置或获取元素的 title 属性

关于Element对象的nodeType、nodeName和nodeValue属性，我们有必要进一步总结一下，首先nodeType会返回一个1~12的数值，每一个值代表一种节点类型，如图7-3所示。

节点类型		描述	子节点
1	Element	代表元素	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
2	Attr	代表属性	Text, EntityReference
3	Text	代表元素或属性中的文本内容	None
4	CDATASection	代表文档中的 CDATA 部分（不会由解析器解析的文本）	None
5	EntityReference	代表实体引用	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
6	Entity	代表实体	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
7	ProcessingInstruction	代表处理指令	None
8	Comment	代表注释	None
9	Document	代表整个文档（DOM 树的根节点）	Element, ProcessingInstruction, Comment, DocumentType
10	DocumentType	向为文档定义的实体提供接口	None
11	DocumentFragment	代表轻量级的 Document 对象，能够容纳文档的某个部分	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
12	Notation	代表 DTD 中声明的符号	None

图 7-3　nodeType 值所表示的类型

nodeName和nodeValue的返回值也会根据nodeType的不同有很大差异，每种节点类型对应的nodeName和nodeValue值如图7-4所示。

节点类型		nodeName 返回	nodeValue 返回
1	Element	元素名	null
2	Attr	属性名称	属性值
3	Text	#text	节点的内容
4	CDATASection	#cdata-section	节点的内容
5	EntityReference	实体引用名称	null
6	Entity	实体名称	null
7	ProcessingInstruction	target	节点的内容
8	Comment	#comment	注释文本
9	Document	#document	null
10	DocumentType	文档类型名称	null
11	DocumentFragment	#document 片段	null
12	Notation	符号名称	null

图 7-4　nodeName 与 nodeValue 值的意义

在学习Element对象中定义的方法之前，建议你将上面列出的常用属性在我们创建的工程框架中编写测试一下。Element对象中定义的方法是操作节点的基础，除了上面的代码示例中演示可以向某个节点内追加子节点，其他常用方法如表7-4所示。

表7-4　Element对象中定义的其他方法

方法名	参数	意义
cloneNode	(deep) 布尔类型，表示是否克隆属性	复制节点
getAttribute	(name) 属性字符串	获取节点某个属性的值
getAttributeNode	(name) 属性字符串	获取节点的某个属性节点
getElementsByTagName	(name) 标签名	获取此节点下所有子节点中标签名为参数指定的字符串的节点集合
hasAttribute	(name) 属性名	将返回布尔值，判断节点是否有某个属性
hasAttributes	无	返回布尔值，判断节点是否有属性

（续表）

方法名	参数	意义
hasChildNodes	无	返回布尔值，判断节点是否有子节点
insertBefore	(new,old) 指定在 old 节点前插入新节点	在指定节点前插入节点
isEqualNode	(node) 节点对象	返回布尔值，判断两个节点对象是否相等 相等的条件： 1. 节点类型相同 2. 拥有相同的节点名、节点值等 3. 子节点相同 4. 拥有相同的属性和属性值
isSameNode	(node) 节点对象	返回布尔值 判断两个节点是否相同
removeAttribute	(name) 属性名字符串	移除元素中的指定属性
removeChild	(node) 节点对象	移除某个子节点
replaceChild	(new,old) 节点对象	替换某个子节点
setAttribute	(name,value) name：属性名 value：属性值	设置元素的属性值
setAttributeNode	(node) 属性节点对象	设置元素属性

7-5　Attribute 属性对象

在网页开发中，HTML代码实际上是页面的框架，比如哪一个地方显示标题、哪一个地方显示内容等，属性则是对框架的内部进行完善与填充，比如标题要显示的精准位置在哪里、文本配置什么样的颜色等。在整个文档树中，HTML标签节点被称为元素节点，属性被称为属性节点，我们可以使用JavaScript来获取具体的属性节点对象来动态操作元素节点的属性。

首先新建一个HTML文件用来作为本示例的测试页面，代码如下：

```
<!DOCTYPE html>
<html>
<head>
```

```
        <title>Attribute 示例</title>
    </head>
    <body>
    <p id="p" class="pClass" style="margin-left: 100px;color:red" myAttribute="Hello World">我是一行文本</p>
    <script type="text/javascript" src="../javaScript/attributeJs.js"></script>
    </body>
    </html>
```

编写JavaScript文件如下：

```
(function(){
    let element = document.getElementById("p");
    let attributes = element.attributes;
    console.log(attributes);      //集合对象，可以通过下标或者属性名来获取具体的属性节点对象
    let a1 = attributes[0];
    console.log(a1.name);         //获取此属性节点的名称
    console.log(a1.value);        //获取此属性节点的值
    //重新设置属性的值
    a1.value = "pp";
    //也可以使用属性名来获取属性节点对象
    let a2 = attributes["myAttribute"];
    console.log(a2);
    let newArrtebute = document.createAttribute("newArrtebute");
    newArrtebute.value = "new";
    //添加新的属性
    attributes.setNamedItem(newArrtebute);
    //删除属性
    attributes.removeNamedItem("style");
})();
```

上面代码中的注释十分清楚，属性分为预设属性和自定义属性，无论哪种属性，我们都可以在attributes集合对象中获取，通过操作Attribute对象的value值，我们可以灵活地改变页面的行为表现。需要注意，setNamedItem和removeNamedItem是定义在属性集合对象上的方法，通过这两个方法可以对Element对象的属性进行新增或删除，Document对象的createAttribute方法可以创建一个新的属性节点。

7-6　用户事件

在网页中，用户操作网页产生的事件被称为Event对象，比如用户对键盘鼠标的操作，对页面元素的选中、单击等都可以作为事件被开发者监听到，通过JavaScript来对用户的行为进行反馈。在HTML元素中可以定义许多事件属性。下面在工程模板中新建一个测试网页，演示一些常用的事件属性。

编写HTML文件如下：

```
<!DOCTYPE html>
<html>
<head>
	<title>Event 示例</title>
	<script type="text/javascript" src="../javaScript/eventJs.js"></script>
</head>
<body>
<div style="margin: 100px">
	<input type="text" name="text" onchange="inputChange(this.value)" />
	<div id="click" onclick="divOnclick(this.id)">点击我</div>
	<div id="dbclick" ondblclick="divOnfbclick(this.id)">双击我</div>
	<input onkeydown="inputonkeydown(event)" value="请按下键盘按键"/>
	<div onmousedown="divonmousedown(event)">鼠标按下</div>
	<div onmousemove="divonmousemove(event)">移动鼠标</div>
</div>
</body>
</html>
```

编写JavaScript文件如下：

```
function inputChange(value){
	alert("输入框修改为"+value);
}
function divOnclick(value){
	alert("点击了"+value);
}
function divOnfbclick(value){
	alert("双击了"+value);
}
```

```
function inputonkeydown(e){
        alert("按下了"+e.key);
}
function divonmousedown(e){
        alert("按下鼠标"+"x:"+e.screenX+" y:"+screenY);
}
function divonmousemove(e){
        alert("移动鼠标"+"x:"+e.screenX+" y:"+screenY);
}
```

在浏览器中运行，可以看到相关效果，需要注意，并不是所有浏览器定义的event对象结构都一致，如果你在运行时有疑惑，可以通过调试模式来看event对象的结构。上面的代码演示的是HTML元素上可以绑定的常用的一些事件属性。表7-5列出了其他常用事件属性。

表7-5　HTML元素上可以绑定的常用的事件属性

事件	备注
Onabort	图像加载中断时触发，只有 img 元素支持
Onblur	元素失去焦点时触发
Onerror	图像加载失败时触发
Onfocus	元素获得焦点时触发
Onkeyup	键盘按键抬起时触发
Onload	页面或图像加载完成时触发
Onkeypress	键盘按钮按下并抬起时触发
Onmouseout	鼠标被移开时触发
Onmouseover	鼠标被移动到某元素上触发
Onmouseup	鼠标按键松开时触发
Onreset	重置按钮被点击时触发
Onresize	窗口大小被调整时触发
Onselect	文本被选中时触发
Onsubmit	确认按钮被单击时触发

7-7　Event 事件对象

上一示例中演示了如何为HTML元素添加事件，演示代码里也涉及event对象，当事件触发后，此事件的行为和相关属性会被封装成event对象传入开发者定义的回调函数，例如键盘按键事件，可以通过这个对象获取用户具体操作的是哪个按钮，如果是鼠标单击事件，那么

可以获取用户单击的是鼠标的哪个按键。表7-6列出了event对象中一些常用的属性，不同浏览器可能会有略微差别。

表7-6　event对象常用属性

属性名	意义
Key	键盘按下的键
keyCode	按键编码
Button	鼠标按键 0：左键 1：中间键 2：右键
clientX	鼠标水平位置
clientY	鼠标垂直位置
ctrlKey	事件触发时 Ctrl 键是否按下，布尔值
altKey	事件触发时 Alt 键是否按下，布尔值
metaKey	事件触发时 Meta 键是否按下，布尔值
screenX	水平坐标
screenY	垂直坐标
shiftKey	事件触发时 Shift 键是否按下
Target	触发此事件的元素
Type	事件名称
Bubbles	获取事件是否是冒泡时间，布尔值

7-8　关于事件传递

HTML文档中的元素往往存在父子关系，我们经常会遇到父元素和子元素都可以接收事件的情况。在浏览器中，处理事件的方式一般有两种：事件冒泡与事件捕获。

事件冒泡是指当用户事件发生时，最内层的子元素先进行处理，然后传递到上层的父元素，一层一层向上传递。事件捕获是指当用户事件发生时，最外层的父元素先进行处理，然后传递到子元素，一层一层向下传递。如果你使用的是Google Chrome浏览器，可以用如下代码来测试事件冒泡：

```
<!DOCTYPE html>
<html>
<head>
    <title>事件冒泡与事件捕获</title>
```

```
</head>
<body>
<div id="div1" onclick="alert(this.id)">
    <div id="div2" onclick="alert(this.id)">
        <div id="div3" onclick="alert(this.id)">
            请点击我
        </div>
    </div>
</div>
</body>
</html>
```

需要注意，并不是所有的场景都需要事件冒泡传递，如果你想让某一个元素处理过事件后不再进行传递，可以调用事件对象的stopPropagation方法。例如，上面的示例代码可以改写如下，这样便只有id为div3的元素会接收到事件：

```
<!DOCTYPE html>
<html>
<head>
    <title>事件冒泡与事件捕获</title>
</head>
<body>
<div id="div1" onclick="alert(this.id)">
    <div id="div2" onclick="alert(this.id)">
        <div id="div3" onclick="alert(this.id);event.stopPropagation()">
            请点击我
        </div>
    </div>
</div>
</body>
</html>
```

7-9　简单的轮播广告

通过对前面知识的学习，你应该已经具备使用JavaScript开发简单网页逻辑的能力。本示例将演示一个简易的轮播广告图的开发过程。

轮播广告是网页中十分常用的一种组件，其通常表现为间隔一段时间后就会变换广告内

容，并且支持用户手动进行广告的切换。首先在项目工程中新建一个src文件夹，将广告素材放入其中，编写HTML文件如下：

```
<!DOCTYPE html>
<html>
<head>
    <title>轮播图</title>
    <style type="text/css">
        #ad{
            width: 100%;
            height: 350px;
        }
        #left{
            width: 30px;
            height: 70px;
            position: absolute;
            top:140px;
        }
        #right{
            width: 30px;
            height: 70px;
            position: absolute;
            right:8px;
            top:140px;
        }
        #container{
            height: :350px;
        }
    </style>
</head>
<body>
<div id="container">
    <img id="left" src="../src/left.png" onclick="left()" />
    <img id="right" src="../src/right.png" onclick="right()" />
    <img id="ad" src="../src/1.jpg" />
<script type="text/javascript" src="../javaScript/sliderAD.js"></script>
</div>
</body>
</html>
```

编写JavaScript文件如下：

```
(function(){
    let adImageElement = document.getElementById("ad");
    document.index = 1;
    setInterval(function(){
        if (document.index>=3) {
            document.index=1;
        }
        adImageElement.setAttribute("src","../src/"+document.index+".jpg");
        document.index++;
    },6000);
})();
function left(){
    let adImageElement = document.getElementById("ad");
    if (document.index>1) {
        document.index--;
        adImageElement.setAttribute("src","../src/"+document.index+".jpg");
    }
}
function right(){
    let adImageElement = document.getElementById("ad");
    if (document.index<3) {
        document.index++;
        adImageElement.setAttribute("src","../src/"+document.index+".jpg");
    }
}
```

在浏览器中运行，可以看到轮播广告图已经可以正常工作。需要注意，我们的轮播广告十分简易，内容的转换也非常生硬，你可以利用互联网查找一下如何使用CSS3来添加过渡动画。

7-10　Window 窗口对象

Window对象用来描述浏览器窗口，使用它可以方便地获取浏览器窗口的相关信息，其中常用属性如表7-7所示。

表7-7　Window对象的常用属性

属性名	意　义
Document	获取页面文档对象
History	获取窗口历史对象
Innerheight	获取页面文档显示区的高度
Innerwidth	获取页面文档显示区的宽度
Location	获取窗口的地址对象
Name	获取窗口的名称
Navigator	获取窗口的导航对象
Opener	获取打开当前窗口的父窗口
Outerheight	获取窗口的外部高度
Outerwidth	获取窗口的外部宽度
pageXOffset	获取当前页面相对于窗口显示区的水平偏移
pageYOffset	获取当前页面相对于窗口显示区的垂直偏移
Parent	获取父窗口
Screen	获取窗口的屏幕对象

表7-8列出了Window对象定义的常用方法，需要注意，这些方法可以使用Window对象来调用，也可以直接调用。

表7-8　Window对象定义的常用方法

方法名	参　数	意　义
Alert	(msg) msg：警告消息字符串	弹出一个警告框
Blur	无	将用户焦点从窗口移开
setInterval	(func,mill) func：回调函数 mill：执行间隔，毫秒	以一定的周期来执行回调函数，会返回定时器 id 值
setTimeout	(func,mill) func：回调函数 mill：延时时间，毫秒	延时一定时间后执行回调函数，会返回定时器 id 值
clearInterval	(id) id：定时器 id 值	取消循环定时器
clearTimeout	(id) id：定时器 id 值	取消延时定时器

（续表）

方法名	参　数	意　义
Close	无	关闭自身窗口，需要注意，这个方法只能关闭由 open 方法打开的窗口
Open	(url,name,fea,rep) url：打开新窗口的地址 name：打开新窗口的名称 fea：窗口特性配置字符串，后边会介绍 rep：布尔值，设置新装载的窗口 URL 是否替换浏览历史中的当前条目	打开一个新的窗口，可以设置其名称和属性
Confirm	(msg) msg：消息字符串	弹出一个确认框，会自带“确认”和“取消”两个按钮，如果用户选择“确认”按钮，会返回 true；如果用户选择“取消”按钮，会返回 false
moveBy	(x,y) x：水平移动像素数 y：垂直移动像素数	将窗口向右下方移动
moveTo	(x,y) x：水平移动像素数 y：水平移动像素数	将窗后移动到指定的位置
Print	无	打印当前页面
Prompt	(text,defaultText) text：对话框标题 defaultText：输入框中的默认文本	弹出一个用户输入框，包含一个“确认”按钮和一个“取消”按钮，如果用户单击“确认”按钮，就会返回用户输入的文字；如果用户单击“取消”按钮，就会返回 null
resizeBy	(width,height) width：增加宽度 height：增加高度	调整窗口尺寸
resizeTo	(width,height) width：宽度 height：高度	调整窗口尺寸到
scrollBy	(x,y) x：水平偏移 y：垂直偏移	将当前页面进行滚动控制
scrollTo	(x,y) x：水平坐标 y：垂直坐标	将当前页面滚动到指定位置

前面有提到，在使用open方法打开窗口时，我们可以为其配置一个属性字符串，例如"width=200,height=200"，这样会将打开的窗口尺寸设置为宽高均为200像素，可配置的属性字段如图7-5所示。

channelmode=yes|no|1|0	是否使用剧院模式显示窗口。默认为 no
directories=yes|no|1|0	是否添加目录按钮。默认为 yes
fullscreen=yes|no|1|0	是否使用全屏模式显示浏览器。默认是 no。处于全屏模式的窗口必须同时处于剧院模式
height=pixels	窗口文档显示区的高度。以像素计
left=pixels	窗口的 x 坐标。以像素计
location=yes|no|1|0	是否显示地址字段。默认是 yes
menubar=yes|no|1|0	是否显示菜单栏。默认是 yes
resizable=yes|no|1|0	窗口是否可调节尺寸。默认是 yes
scrollbars=yes|no|1|0	是否显示滚动条。默认是 yes
status=yes|no|1|0	是否添加状态栏。默认是 yes
titlebar=yes|no|1|0	是否显示标题栏。默认是 yes
toolbar=yes|no|1|0	是否显示浏览器的工具栏。默认是 yes
top=pixels	窗口的 y 坐标
width=pixels	窗口的文档显示区的宽度。以像素计

图 7-5　打开窗口可配置的窗口属性字段

7-11　Navigator 导航对象

Navigator导航对象中包含浏览器的相关信息，使用它可以获取浏览器的名称、版本、平台等信息。表7-9列出了Navigator对象中的常用属性。

表7-9　Navigator对象中的常用属性

属性名	意　义
appCodeName	获取浏览器的代码名
appName	获取浏览器的名称
appVersion	获取浏览器平台和版本信息
cookieEnabled	获取浏览器是否开启 Cookie 支持
onLine	获取浏览器当前是否是非脱机模式
Platform	获取浏览器当前运行的操作系统平台
userAgent	获取浏览器用户请求的用户代理头的值

如果要检查当前浏览器是否启用Java，可以使用JavaEnabled函数，如果返回true，表示支持；如果返回false，表示不支持。

7-12　Screen 屏幕对象

Screen对象中包含显示屏的相关信息，通过它可以更加方便地对新窗口进行定位，也可以根据屏幕的分辨率选择合适的图片显示给用户。其中，常用属性如表7-10所示。

表7-10　Screen的常用属性

属性名	意　义
availHeight	获取可显示区域高度
availWidth	获取可显示区域宽度
colorDepth	获取颜色比特深度
height	获取显示屏的高度
width	获取显示屏的宽度
pixelDepth	获取颜色分辨率

7-13　History 历史对象

History对象中包含用户在当前窗口所访问过的URL历史，并且开发者可以使用这个对象来控制页面的前进与后退。History对象中有一个名为length的属性，这个属性可以获取当前窗口历史中有多少个URL。需要注意，出于隐私考虑，History对象并不能直接获取用户所加载过的URL，只能使用相关方法来控制页面的跳转。表7-11列出了History对象的常用方法。

表7-11　History对象的常用方法

方法名	参　数	意　义
Go	(num|url) 参数可以是一个整数值，也可以是一个 URL 字符串，如果是整数值，就表示跳转到相对当前位置偏移参数数值的某个 URL，例如传入-1，表示跳转到上一个 URL 页面。若传入 URL 字符串，则会直接跳转到此 URL	进行历史页面跳转

（续表）

方法名	参　数	意　义
Back	无	加载前一个 URL（如果存在）
Forward	无	加载后一个 URL（如果存在）

7-14　Location 地址对象

Location对象是对当前URL信息的封装，并且提供了方法用来重新加载或者替换当前页面的URL。其中，常用属性如表7-12所示。

表7-12　Location对象的常用属性

属性名	意　义
Host	设置或返回当前页面的主机名和端口号，如果是设置，就会重新加载页面
Hostname	设置或返回当前的主机名
Href	设置或返回当前页面的完整 URL
Pathname	设置或返回当前 URL 的路径部分
Port	设置或返回当前 URL 的端口号
Protocol	设置或返回当前 URL 的协议
Search	设置或返回当前 URL 的参数部分

表7-13给出了Location对象几个常用的加载URL的方法。

表7-13　Location常用的加载URL的方法

方法名	参　数	意　义
Assign	(url)	加载新的文档
Reload	无	重新加载当前文档
Replace	(url)	加载新的文档，与 Assign 方法不同的是，这个方法不会在 History 中生成新的记录，只会替换当前的 URL

7-15　编程练习

练习1：给定一个ul列表，单击每个列表中的li项删除此li项。

解析：

参考代码如下：

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title></title>
	</head>
	<body>
		<ul id="list">
			<li>1</li>
			<li>2</li>
		<li>3</li>
		<li>4</li>
		<li>5</li>
		</ul>
	</body>
		<script type="application/javascript">
			window.onload = function() {
		var list = document.getElementsByTagName("ul")[0].childNodes;

		for (var i = 0; i < list.length; i++) {
			list[i].onclick = function() {

				this.parentNode.removeChild(this);
			}
		}
	}
	</script>
</html>
```

练习2：给定一个ul列表，单击其中最后一个li元素后动态追加li。

解析：

参考代码如下：

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title></title>
	</head>
	<body>
```

```
            <ul id="list">
                <li>1</li>
                <li>2</li>
              <li>3</li>
              <li>4</li>
              <li>5</li>
            </ul>
        </body>
                <script type="application/javascript">
                        window.onload = function() {
                    var list = document.getElementsByTagName("li");
                   var li = list[list.length-1];
                    li.onclick = function func() {
                     var lii = document.createElement('li');
                     lii.innerHTML = list.length+1;
                     lii.onclick = func;
                       this.parentNode.appendChild(lii);
                    }
              }
          </script>
</html>
```

练习3：给定一个ul列表，单击其中某个li元素后，和其下一个元素进行交换，如果没有下一个元素，就和上一个元素进行交换。

解析：

参考代码如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
            <ul id="list">
                <li>1</li>
                <li>2</li>
              <li>3</li>
              <li>4</li>
              <li>5</li>
```

```
            </ul>
        </body>
                <script type="application/javascript">
                        window.onload = function() {
                    var list = document.getElementsByTagName("li");
                    var l = document.getElementById('list');
                    for (let i= 0;i<list.length;i++) {
                      list[i].onclick = function func() {
                          if (i == list.length-1){
                                l.insertBefore(list[i],list[i-1]);
                          }else{
                                l.insertBefore(list[i+1],list[i]);
                          }

                     }
                    }

            }
          </script>
</html>
```

第 8 章

JavaScript 项目实战

通过前面几个章节的学习，相信你已经掌握了JavaScript语言的核心语法，并且有了操作HTML元素的能力。本章将更多地综合运用前面所学习的知识进行两个简单有趣的小项目的实战开发。通过学习本章内容，你的JavaScript运用能力将会得到进一步提高。

8-1　项目一：编写一个简易网页时钟

你一定见过各式各样的电子时钟，有没有想过自己来开发一款呢？其实编写一个简易的网页时钟应用十分简单。例如，HTML 5的Canvas标签可以随心所欲地在网页上绘制自定义的图形。本实战项目将教你如何一步一步地实现一款简易的网页时钟，实现的效果如图8-1所示。

需要注意，我们要实现的时钟并不是静态的，会根据当前的系统时间实时走动。我们只需要使用定时器进行刷新绘制即可实现。

简易时钟

Wed Mar 28 2018

图 8-1　简易网页时钟效果

8-1-1　关于 Canvas 标签

在编写时钟界面前，你首先要对Canvas标签有个初步的了解。HTML 5中提供了<canvas>标签来进行图像的绘制，当然需要通过JavaScript脚本来进行操作。

Canvas相当于一张画布，当你需要向画布上绘制内容时，需要先获取绘制上下文对象，例如在界面上画出一个矩形图形，可以使用如下代码：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
        <canvas id='canvas'></canvas>
    </body>
    <script type="application/javascript">
        let context = document.getElementById('canvas').getContext("2d");
        context.rect(30,30,100,100);
        context.fillStyle = 'red'
        context.fill();
    </script>
</html>
```

上面的代码中，getContext("2d")用来获取绘制上下文；rect()函数用来添加一个矩形图形，

其中4个参数分别对应矩形的x坐标、y坐标、宽度和高度；fillStyle属性用来设置填充的风格，可以用来设置绘制的颜色；fill()函数用来对添加到绘制上下文中的图形进行渲染绘制。

在绘制图形时，除了可以设置颜色外，还有许多属性可以供我们使用，例如阴影、边框、模糊等效果，示例代码如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
          <canvas id='canvas'></canvas>
    </body>
    <script type="application/javascript">
          let context = document.getElementById('canvas').getContext("2d");
          context.rect(30,30,100,100);
          //设置线条颜色
          context.strokeStyle = 'blue';
          //设置填充颜色
          context.fillStyle = 'red';
          //设置阴影颜色
          context.shadowColor = 'green';
          //设置阴影 X 偏移
          context.shadowOffsetX = 10;
          //设置阴影 Y 偏移
          context.shadowOffsetY = 10;
          //设置线条宽度
          context.lineWidth = 3;
          //进行填充绘制
          context.fill();
          //进行边框绘制
          context.stroke();
    </script>
</html>
```

绘制效果如图8-2所示。



图 8-2　图形绘制效果

除了矩形外，你也可以绘制任意路径的图形，beginPath()、moveTo()和closePath()3个函数用来绘制闭合图形，例如绘制三角形的核心代码如下：

```
context.beginPath();
context.moveTo(30,30);
context.lineTo(30,150);
context.lineTo(100,150);
context.closePath();
context.stroke();
```

弧线和圆的绘制方法也十分简单，核心代码如下：

```
context.arc(100,100,80,0,Math.PI*2,false);
context.fill();
```

arc()函数中的6个参数分别用来设置圆心x坐标、圆心y坐标、半径、起点弧度值、终点弧度值和是否逆时针绘制。上面代码的绘制效果如图8-3所示。

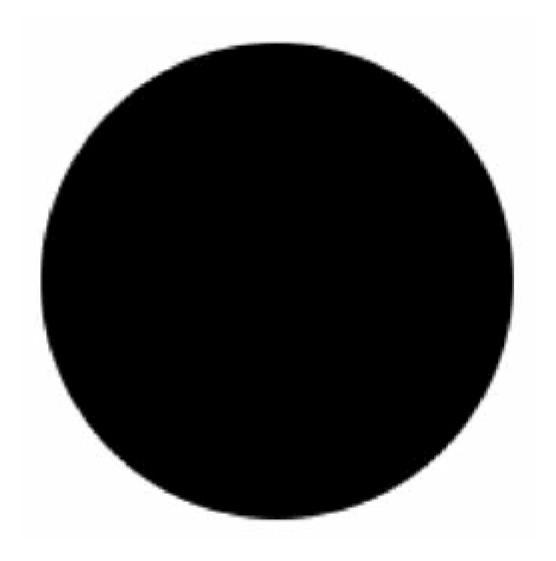


图 8-3　圆形绘制效果

关于文本的绘制可以采用fillText()或者strokeText()函数，这里就不再一一列举。使用Canvas也可以十分轻松地绘制出渐变图形，在后面的时钟实战中会有应用。

8-1-2　制作简易网页时钟

网页时钟的参考代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>时钟</title>
</head>
<body>
    <div style="text-align: center;"><h1>简易时钟</h1></div>
    <div style="text-align: center;"><canvas id="one" width="800" height="500"></canvas></div>
    <div style="text-align: center; font-size: 35px;" id="date"></div>
    <script>
        var date = document.getElementById("date");
        var d = new Date();
        //获取当前日期字符串
        date.innerText = d.toDateString();
      var one=document.getElementById("one");
      var context=one.getContext("2d");
      function clock() {
        context.clearRect(0,0,800,800);         //清除画布
            //获取当前时间
          var sec=new Date().getSeconds();
          var min=new Date().getMinutes();
          var hour=new Date().getHours();
          //设置表盘
          //圆形渐变色参数分别为两个圆心和半径
          var g1 = context.createRadialGradient(400, 250, 0, 400, 250, 200);
          //设置两个渐变色的参数
          g1.addColorStop(0, "#fcfcfc");
          g1.addColorStop(1, "#defdff");
          //设置线宽
          context.lineWidth = 2;
          context.fillStyle = g1;
          context.beginPath();
          //绘制圆盘
          context.arc(400, 250, 200, 0,360, false);
          context.closePath();
          context.fill();
          context.stroke();
          context.fillStyle="yellow";
```

```
context.beginPath();
//绘制表芯
context.arc(400,250,10,0,360, false);
context.closePath();
context.fill();
//时针刻度盘
for (var i = 0; i < 12; i++) {
    context.save();
    //将坐标原点位置移动到圆心处
    context.translate(400, 250);
    //将表盘旋转
    context.rotate(i*30*Math.PI/180);
    context.beginPath();
    context.moveTo(0, -200);
    context.lineTo(0, -180);
    context.closePath();
    context.stroke();
    context.restore();
}
//分针刻度盘和秒针刻度盘
for (var i = 0; i < 60; i++) {
    if (i%5!=0) {
        context.save();
        context.strokeStyle="red";
        context.translate(400, 250);
        //旋转表盘
        context.rotate(i*6*Math.PI/180);
        context.beginPath();
        context.moveTo(0, -200);
        context.lineTo(0, -190);
        context.closePath();
        context.stroke();
        context.restore();
    }
}
context.save();
context.fillStyle="black";
  var hours=[3,4,5,6,7,8,9,10,11,12,1,2];          //定义时间数组
  context.font="26px pxArial"                      //字体和字体大小
```

```
        context.textAlign='center';                     //数字相对圆心左右两边居中
        context.textBaseline='middle';                  //数字相对圆心上线两边居中
        //画时刻
        hours.forEach(function(number,i){
        var rad=2*Math.PI/12*i;                         //把圆分为 12 个弧度
        var x=Math.cos(rad)*(200-30)+400;
        var y=Math.sin(rad)*(200-30)+250;
    context.fillText(number,x,y);
  });
  context.stroke();
  context.restore();
    //秒针
    context.save();
    context.fillStyle="red";
    context.translate(400, 250);
    context.rotate(sec*6*Math.PI/180);
    context.beginPath();
    context.moveTo(0, -170);
    context.lineTo(-5, -30);
    context.lineTo(0, -10);
    context.lineTo(5, -30);
    context.closePath();
    context.fill();
    context.restore();
    //分针
    context.save();
    context.fillStyle="blue";
    context.translate(400,250);
    context.rotate(min*6*Math.PI/180);
    context.beginPath();
    context.moveTo(0, -150);
    context.lineTo(-5, -30);
    context.lineTo(0, -10);
    context.lineTo(5, -30);
    context.closePath();
    context.fill();
    context.restore();
    //时针
    context.save();
```

```
                context.lineWidth=2;
                context.fillStyle="black";
                context.translate(400,250);
                context.rotate(hour*30*Math.PI/180);
                context.beginPath();
                context.moveTo(0, -130);
                context.lineTo(-8,-30);
                context.lineTo(0,-10);
                context.lineTo(8,-30);
                context.closePath();
                context.fill();
                context.restore();
            }
            clock();
            setInterval(clock,1000);
        </script>
    </body>
    </html>
```

上面的代码有比较详尽的注释，本实战练习的编写难度并不大，主要是一些简单的数学角度计算和使用JavaScript对Canvas的操作。但是实现的效果是十分炫酷的，如果你觉得JavaScript只能用来做一些界面和动画，那你就错了。下一个实战项目将一起来实现一个网页笑话阅读器。

8-2　项目二：编写网页笑话阅读器

本实战项目将开发一款简单的网页阅读器App。使用JavaScript开发网页的一个非常大的优势是将以前复杂的服务端逻辑分担在了客户端。Ajax技术结合JavaScript脚本可以开发出体验十分优秀的网页应用。

在前面的章节中，我们一直没有使用网络技术，本节将通过网络请求获取有趣的图文资源来进行展示，因此，在开始之前，我们需要一个服务端来提供实时更新的资源。

8-2-1　通过互联网获取免费的应用数据

互联网上有许多数据供应商会提供免费的数据，我们可以使用这些数据来练习与学习。易源数据网是一个很不错的数据交易平台，其网址为：https://www.showapi.com/，如果你没有易源数据网的会员账号，需要先注册一下，注册的过程是免费的。

注册完账号，登录成功后，你可以在首页的搜索栏中输入“笑话”来搜索本实战应用需要使用的数据服务，如图8-4所示。



图 8-4　进行数据服务搜索

你会在搜索结果中找到“笑话大全”服务，它就是我们所需要的接口服务，并且可以免费使用，如图8-5所示。



图 8-5　使用“笑话大全”服务

进入笑话大全服务，单击界面上的“订购”，之后即可使用此服务的接口。

“笑话大全”服务中提供了3个接口，分别用来获取文本类笑话、图片笑话和动态图笑话，本实战只需要使用前两个接口。

接口服务详情页里面有详情的参数文档和返回数据的数据结构文档，参数文档如图8-6所示。

1、系统级参数（所有接入点都需要的参数）：收起

参数名称	类型	示例值	必须	描述
showapi_appid	String	100	是	易源应用id
showapi_sign	String	698d51a19d8a121ce581499d7b701668	是	为了验证用户身份，以及确保参数不被中间人篡改，需要传递调用者的数字签名。
showapi_timestamp	String	20141114142239	否	客户端时间。 格式yyyyMMddHHmmss,如20141114142239 为了在一定程度上防止"重放攻击"，平台只接受在10分钟之内的请求。如果不传或传空串，则系统不再做此字段的检测。
showapi_sign_method	String	md5	否	签名生成方式，其值可选为"md5"或"hmac"。如果不传入则默认"md5"。
showapi_res_gzip	String	1或0	否	返回值是否用gzip方式压缩。此值为1时将压缩，其他值不压缩。

2、应用级参数（每个接入点有自己的参数）：

参数名称	类型	默认值	示例值	必须	描述
page	String	1	1	否	第几页。
maxResult	String	20	20	否	每页最大记录数。其值为1至50

图 8-6　接口服务的参数文档

从文档中可以看到，showapi_appid和showapi_sign两个参数是必传的，这两个参数用来进行请求者身份的验证。在易源数据网的个人中心中，你可以查看自己的appId和sign密钥，如图8-7所示。



图 8-7　查看自己的 appId 和密钥的值

下面可以测试一下，打开浏览器，在地址栏中输入如下地址：

http://route.showapi.com/341-2?showapi_appid=58027&showapi_sign=74b9fcd59b844b98b6427da974f4e2e9

需要注意，上面的地址拼接的showapi_appid和showapi_sign的值需要换成你自己在个人中心查找到的相关值。访问这个地址，如果有数据返回，如图8-8所示，就说明你的数据服务已经可以使用。

route.showapi.com/341-2?showapi_appid=58027&showapi_sign=74b9fcd59b844b98b6427da974f4e2e9

{"showapi_res_error":"","showapi_res_code":0,"showapi_res_body":{"allPages":34,"ret_code":0,"contentlist":[{"title":"妹子们在聊什么","img":"http://sc2.hao123img.com/data/2d8c341e48810bebd7bbf4ea43a5705e_430","type":2,"ct":"2018-03-28 15:30:02.184"},{"title":"搞笑图片第1487期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180328/12491542D-0.jpg","type":2,"ct":"2018-03-28 13:00:05.226"},{"title":"搞笑图片第1489期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180328/1250501006-0.jpg","type":2,"ct":"2018-03-28 13:00:05.226"},{"title":"搞笑图片第1490期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180328/125240L58-0.jpg","type":2,"ct":"2018-03-28 13:00:05.226"},{"title":"搞笑图片第1486期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180328/124Q544D-0.jpg","type":2,"ct":"2018-03-28 13:00:05.224"},{"title":"搞笑图片第1488期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180328/1249525015-0.jpg","type":2,"ct":"2018-03-28 13:00:05.224"},{"title":"各个年龄通用！","img":"http://sc2.hao123img.com/data/17371b2501b5bac7ae574ae1843c45c6_430","type":2,"ct":"2018-03-27 20:00:02.252"},{"title":"妹子别挡脸啊","img":"http://sc4.hao123img.com/data/7a707fcf180918be54a68db87ed2d85c_430","type":2,"ct":"2018-03-27 19:30:04.537"},{"title":"这么晒香肠，味道不会怪怪的吗？","img":"http://sc4.hao123img.com/data/f289813e0236b653bea54b06e8016de9_430","type":2,"ct":"2018-03-27 18:00:02.584"},{"title":"妹子也太懒了","img":"http://sc1.hao123img.com/data/eaf2181ccac6c35fbd0c74d24b727562_430","type":2,"ct":"2018-03-27 17:30:02.830"},{"title":"搞笑图片第1476期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180327/100Q03609-0.jpg","type":2,"ct":"2018-03-27 10:30:10.350"},{"title":"搞笑图片第1477期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180327/100U04Z8-0.jpg","type":2,"ct":"2018-03-27 10:30:10.344"},{"title":"搞笑图片第1478期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180327/1009254039-0.jpg","type":2,"ct":"2018-03-27 10:30:10.235"},{"title":"搞笑图片第1479期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180327/1010056320-0.jpg","type":2,"ct":"2018-03-27 10:30:10.213"},{"title":"搞笑图片第1480期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180327/1010431G1-0.jpg","type":2,"ct":"2018-03-27 10:30:10.195"},{"title":"这个表情的宝宝我还是头一次见","img":"http://sc0.hao123img.com/data/070fbc291309d144fe55f24c43b3dadf_0","type":2,"ct":"2018-03-27 00:30:01.654"},{"title":"妹子一看就是练过的","img":"http://sc4.hao123img.com/data/9d5872e774d07565d9f4bd164d7beb74_430","type":2,"ct":"2018-03-26 22:30:02.088"},{"title":"大学校园看到的。。。无法反驳","img":"http://sc3.hao123img.com/data/4b32d688efa77a531ef6d7d4747651dd_430","type":2,"ct":"2018-03-26 22:00:03.388"},{"title":"搞笑图片第1466期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180326/131PR018-0.jpg","type":2,"ct":"2018-03-26 13:30:04.888"},{"title":"搞笑图片第1467期（共5图）","img":"http://www.zbjuran.com/uploads/allimg/180326/1320256445-0.jpg","type":2,"ct":"2018-03-26 13:30:04.888"}],"currentPage":1,"allNum":677,"maxResult":20}}

图 8-8　返回数据示例

8-2-2　关于 AJAX

学习JavaScript编程，AJAX是必须学习的一种技术。AJAX全称为Asynchronous JavaScript and XML，也可以解释为异步的JavaScript和XML技术。它是一种在不重新加载整个页面的情况下与服务器交换数据的技术，配合JavaScript HTML DOM技术可以实现实时的网页局部加载或更新，使得网页更加动态，用户体验更加精致。

AJAX技术的核心在于XMLHttpRequest对象，它是HTTP请求对象，使用它来进行请求的配置、发送、数据的接收等。例如，我们使用AJAX来获取上面的接口的数据，代码示例如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
    <script type="application/javascript">
```

```
            var request = new XMLHttpRequest();
            request.open("GET","http://route.showapi.com/341-2?showapi_appid=
58027&showapi_sign= 74b9fcd59b844b98b6427da974f4e2e9",true);
            request.send();
        </script>
    </html>
```

open()函数用来打开一个请求，其第1个参数设置请求的方法，常用的有“GET”和“POST”方法；第2个参数设置请求的URL；第3个参数设置是否异步请求，如果设置为true，那么请求的进行不会阻塞后面JavaScript脚本代码的执行，如果设置为false，那么只有请求结果返回，脚本代码才会向后执行。最后，不要忘了调用send()函数来进行请求的发送。

在Chrome浏览器中运行代码，打开开发者工具，在其中的网络模块可以看到请求的发送和接收到的数据，如图8-9所示。

下面使用XMLHttpRequest对象的回调函数来接收获取到的数据，修改代码如下：

```
    <!DOCTYPE html>
    <html>
        <head>
            <meta charset="UTF-8">
            <title></title>
        </head>
        <body>
        </body>
        <script type="application/javascript">
            var request = new XMLHttpRequest();
            request.onreadystatechange = function(){
                if(request.readyState==4 && request.status==200){
                    document.body.innerText = request.responseText;
                }
            }
            request.open("GET","http://route.showapi.com/341-2?showapi_appid=
58027&showapi_sign= 74b9fcd59b844b98b6427da974f4e2e9",true);
            request.send();
        </script>
    </html>
```

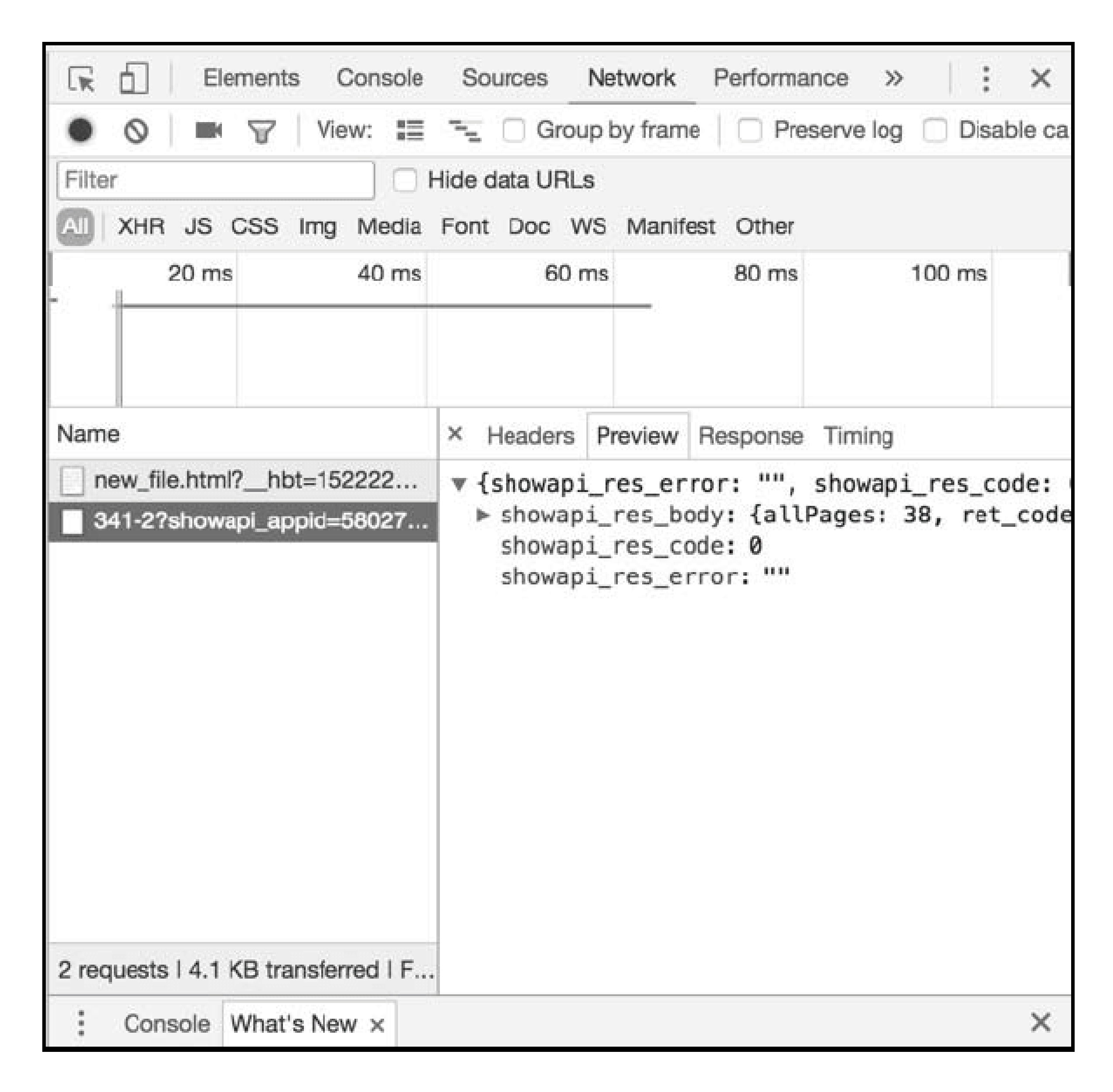


图 8-9　在开发者工具中查看网络请求

onreadystatechange用来设置一个回调函数，当请求的状态发生改变时，此函数会被调用。XMLHttpRequest请求对象的readyState属性可以获取当前请求的状态，其状态码与意义如表8-1所示。

表8-1　状态码及其意义

状态码	意　义
0	请求未初始化
1	服务器连接已建立
2	请求已接收
3	请求处理中
4	请求已完成

XMLHttpRequest请求对象的status属性用来表示服务端做出的响应状态，通常情况下，200表示响应成功。XMLHttpRequest请求对象的responseText为服务端返回的文本数据。上面的代码将请求的数据获取到，并填充进文档的body标签中。

8-2-3　代码实现

下面我们来看看AJAX的应用和使用JavaScript动态操作网页。首先新建一个工程文件夹，将其命名为Laugh，在其中新建3个文件夹，分别命名为css、js和img，这些文件夹用来放置样式表、JavaScript脚本和图片素材，并在根目录Laugh文件夹中新建一个index.html文件。

首先编写index.html文件，在其中进行应用架构的搭建，参考代码如下：

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8" />
		<title></title>
		<link rel="stylesheet" type="text/css" href="css/index.css"/>
		<script type="text/javascript" src="js/index.js" ></script>
	</head>
	<body class="body">
		<h1 class="title">笑一笑，十年少</h1>
		<div class="button">趣图吧</div>
		<div class="container" id="container">
		</div>
		<div class="container" id="container2" hidden="hidden">
		</div>
	</body>
</html>
```

上面的第1个div标签作为切换按钮，用来切换文本笑话和图片笑话模式。通过为div标签添加hidden属性来控制模块的显示和隐藏。

在js文件夹下新建一个index.js文件，在其中编写如下代码：

```
(function(window){
	//默认从第 1 页开始请求数据
	//文本笑话页数
	window.Tpage = 1;
	//图片笑话页数
	window.Ipage = 1;
	//为窗口添加加载完成的回调
	window.onload = function(){
		let button = document.getElementsByClassName('button')[0];
		//为按钮添加点击事件
		button.addEventListener('click',function(){
			if(button.innerText == '趣图吧'){
				button.innerText = "笑一笑";
				let e = document.getElementById("container");
				let e2 = document.getElementById("container2");
				e.setAttribute('hidden','hidden');
```

```
                        e2.removeAttribute('hidden');
                    }else{
                        button.innerText = "趣图吧";
                        let e = document.getElementById("container");
                        let e2 = document.getElementById("container2");
                        e2.setAttribute('hidden','hidden');
                        e.removeAttribute('hidden');
                    }
                });
                //请求第一页数据
                requestData(0);
                requestData(1);
            }
            //监听窗口的滚动，当滚动到底部时，进行加载更多的操作
            window.onscroll = function(event){
                var wScrollY = window.scrollY; // 当前滚动条位置
                var wInnerH = window.innerHeight; // 设备窗口的高度
              var bScrollH = document.body.scrollHeight; // 滚动条总高度
              if (wScrollY + wInnerH >= bScrollH) {
                  //进行加载更多
                  let button = document.getElementsByClassName('button')[0];
                  if(button.innerText=="笑一笑"){
                        requestData(1);
                  }else{
                        requestData(0);
                  }

              }
            }
        })(window);
        //请求数据函数
        function requestData(type){
                var request = new XMLHttpRequest();
                if(type==1){
                    page = window.Ipage++;
                    request.open("GET"," http://route.showapi.com/341-2?showapi_appid=
58027&showapi_sign=74b9fcd59b844b98b6427da974f4e2e9&page="+page,true);
                }else{
                    page = window.Tpage++;
```

```
                    request.open("GET","http://route.showapi.com/341-1?showapi_appid=
58027&showapi_sign=74b9fcd59b844b98b6427da974f4e2e9&page="+page,true);
                }
                request.onreadystatechange = function(){
                    if (request.readyState==4 && request.status==200)
                        {
                            var data = JSON.parse(request.responseText);
                            for (let i = 0;i<data.showapi_res_body.contentlist.length;i++) {
                                if(type==0){
                                    let ele = createElement(data.showapi_res_body.contentlist[i]);
                                    let con = document.getElementById('container');
                                    con.appendChild(ele);
                                }else{
                                    let ele = createElementImg(data.showapi_res_body.contentlist[i]);
                                    let con = document.getElementById('container2');
                                    con.appendChild(ele);
                                }

                            }
                        }
                    }
                request.send();
        }
        //创建文本 div 区域
        function createElement(data){
            var div = document.createElement('div');
            var tit = document.createElement('div');
            var content = document.createElement('div');
            div.appendChild(tit);
            div.appendChild(content);
            tit.innerHTML = data.title;
            content.innerHTML = data.text;
            tit.setAttribute('class','cTitle');
            div.setAttribute("class","cContanier");
            content.setAttribute("class","cContent");
            return div;
        }
        //创建图片 div 区域
        function createElementImg(data){
```

```
        var div = document.createElement('div');
        var tit = document.createElement('div');
        var content = document.createElement('img');
        div.appendChild(tit);
        div.appendChild(content);
        tit.innerHTML = data.title;
        content.innerHTML = data.text;
        tit.setAttribute('class','cTitle');
        div.setAttribute("class","cContanier");
        content.setAttribute("width","360px");
        content.setAttribute('src',data.img);
        return div;
    }
```

上面的代码就是我们整个应用的核心逻辑部分，还需要添加一个样式表文件来进行界面的布局控制，在css文件夹下新建一个命名为index.css的文件，在其中添加如下代码：

```
    .body{
        background: #eeeeee;
    }
    .title{
        background: #eeeeee;
        text-align: center;
        position: fixed;
        width: 100%;
        margin: 0 auto;
        top: 0px;
        padding-top: 20px;
    }
    .container{
        background: white;
        width: 400px;
        margin: 0 auto;
        margin-top: 80px;
    }
    .button{
        position: fixed;
        background: blue;
        color: white;
        border-radius: 5px;
```

```
        width: 110px;
        text-align: center;
        height: 40px;
        line-height: 40px;
        top: 20px;
}
.cTitle{
        text-align: center;
        padding: 20px;
        font-size: 25px;
}
.cContanier{
        padding-bottom: 20px;
        border-bottom: dashed 1px;
        padding-left: 20px;
        padding-right: 20px;
}
.cContent{
        text-indent: 30px;
}
```

在浏览器中运行项目，效果如图8-10与图8-11所示。



图 8-10　文本笑话界面



图 8-11　图片笑话界面